

版权注意事项：

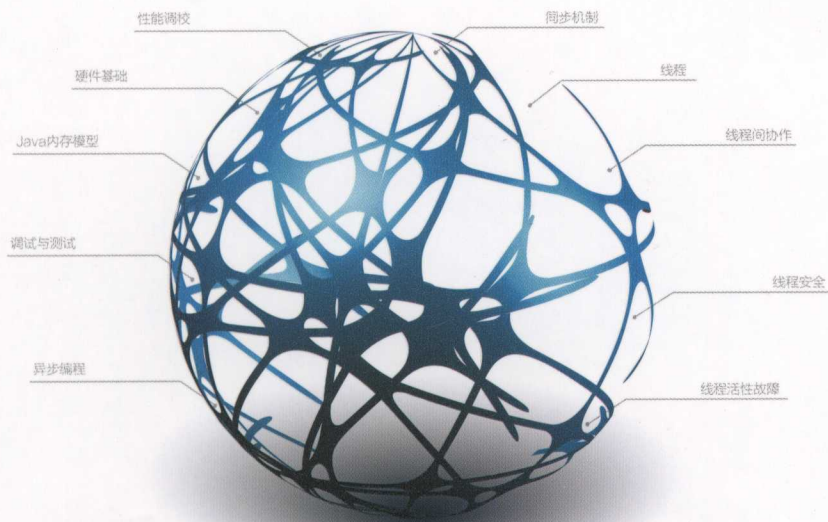
- 1、书籍版权归作者和出版社所有
- 2、本PDF仅限用于个人获取知识，进行私底下的知识交流
- 3、PDF获得者不得在互联网上以任何目的进行传播
- 4、如觉得书籍内容很赞，请购买正版实体书，支持作者
- 5、请于下载PDF后24小时内删除本PDF。

本书以基本概念、原理与方法为主线,辅以丰富的实战案例和生活化实例,
从 Java 虚拟机、操作系统和硬件多个层次与角度出发循序渐进、系统地介绍
Java 平台下的多线程编程核心技术及相关工具。

Java 多线程编程 实战指南

(核心篇)

黄文海 / 著



中国工信出版集团



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

Java多线程编程 实战指南

(核心篇)

黄文海 / 著

电子工业出版社

Publishing House of Electronics Industry

北京·BEIJING

内 容 简 介

随着现代处理器的生产工艺从提升处理器主频频率转向多核化,即在一块芯片上集成多个处理器内核(Core),多核处理器(Multicore Processor)离我们越来越近了——如今就连智能手机这样的消费类设备都已配备了4核乃至8核的处理器,更何况商用系统!在此背景下,以往靠单个处理器自身处理能力的提升所带来的软件计算性能提升的那种“免费午餐”已不复存在,这使得多线程编程在充分利用计算资源、提高软件服务质量方面扮演了越来越重要的角色。故而,掌握多线程编程技能对广大开发人员的重要性亦由此可见一斑。本书以基本概念、原理与方法为主线,辅以丰富的实战案例和生活化实例,并从Java虚拟机、操作系统和硬件多个层次与角度出发,循序渐进、系统地介绍Java平台下的多线程编程核心技术及相关工具。

本书适合有一定Java语言基础的读者作为入门多线程编程之用,也适合有一定多线程编程经验的读者作为重新梳理知识结构以提升认知层次和参考之用。

未经许可,不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有,侵权必究。

图书在版编目(CIP)数据

Java多线程编程实战指南.核心篇 / 黄文海著. —北京:电子工业出版社, 2017.4

(Java多线程编程实战系列)

ISBN 978-7-121-31065-2

I. ①J… II. ①黄… III. ①JAVA语言—程序设计—指南 IV. ①TP312.8-62

中国版本图书馆CIP数据核字(2017)第047559号

策划编辑:付 睿

责任编辑:李云静

印 刷:三河市良远印务有限公司

装 订:三河市良远印务有限公司

出版发行:电子工业出版社

北京市海淀区万寿路173信箱 邮编:100036

开 本:787×980 1/16 印张:30 字数:618千字

版 次:2017年4月第1版

印 次:2017年4月第1次印刷

印 数:3000册 定价:89.00元

凡所购买电子工业出版社图书有缺损问题,请向购买书店调换。若书店售缺,请与本社发行部联系,联系及邮购电话:(010)88254888, 88258888。

质量投诉请发邮件至 zltz@phei.com.cn, 盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式:010-51260888-819, faq@phei.com.cn。

前言

随着现代处理器的生产工艺从提升处理器主频频率转向多核化,即在一块芯片上集成多个处理器内核(Core),多核处理器(Multicore Processor)离我们越来越近了——如今就连智能手机这样的消费类设备都已配备了4核乃至8核的处理器,更何况商用系统!在此背景下,以往靠单个处理器自身处理能力的提升所带来的软件计算性能提升的那种“免费午餐”已不复存在,这使得多线程编程在充分利用计算资源、提高软件服务质量方面扮演了越来越重要的角色。故而,掌握多线程编程技能对广大开发人员的重要性亦由此可见一斑。

本书内容及特色

本书以基本概念、原理与方法为主线,辅以丰富的实战案例和生活化实例,并从Java虚拟机、操作系统和硬件多个层次与角度出发,循序渐进、系统地介绍Java平台下的多线程编程核心技术及相关工具。

脉络清晰、循序渐进和系统性介绍。全书围绕多线程编程所要解决的问题(所要实现的目标)及其面临的各种挑战,由此展开介绍多线程编程中的相关概念、原理与技术。本书以先介绍相关问题及背景再给出相应的解决方案的方式来讲解新的概念、知识。并且,本书对概念、原理与技术的讲解会适当地深入到Java虚拟机、操作系统和硬件这三个层次与角度,而不仅仅停留在Java语言层面。全书章节是按照知识间的内在联系并依照认知程度上的由浅至深的顺序组织的。

以基本概念、原理与方法为主线。本书既注重实战又注重理论对实践的指导作用。本书以多线程编程的基本概念、原理与方法为主线,将Java平台中与多线程编程相关的关键字、Java标准库类(API)等知识串在其上进行讲解,并在讲解过程中适当穿插相关工具的介绍。本书在介绍相关Java标准库类时,不仅仅介绍其API,还适当介绍其内部实现

原理与实战注意事项。

辅以丰富的实战案例和生活化实例。本书配有丰富的实战案例，这些案例的配套源码都是可以实际运行的，以便读者进行实验。本书在介绍一些概念和原理的时候往往辅以一些生活化实例以增加读者的感性认识，降低理解难度。

答疑解惑。本书讲解过程中会穿插一些“扩展阅读”的内容，这部分内容以问答的形式来对多线程编程的初学者在学习和工作过程中经常遇到的一些疑惑和问题进行解答。

本书一共分为 12 章，各章的主要内容如下。

第 1 章主要介绍线程及多线程编程这两个基本概念，以及 Java 平台的线程 API，并通过一个实战案例使读者对多线程编程有个初步和感性的认识。

第 2 章主要介绍多线程编程所要实现的目标及其面临的挑战。明确多线程编程的目标有助于我们在实践中做到有的放矢，掌握多线程编程所面临的挑战使得我们在学习本书后续内容时能够做到知其然而且知其所以然。

第 3 章主要介绍 Java 平台所提供的能够用于应对多线程编程所面临的部分挑战的一些关键字和标准库类（API），以及这些关键字和 API 的性能开销、适用场景及注意事项。

第 4 章通过实战案例介绍具体如何玩转线程以实现多线程编程的目标，并通过这些实战案例展开介绍多线程编程实践中的一些实际问题及应对措施。

第 5 章主要介绍线程与线程之间通过哪些常见的协作形式来实现多线程编程的目标以及 Java 所支持的相应标准库类。

第 6 章主要从软件设计的角度出发介绍应对多线程编程所面临的一些挑战的常见技术。

第 7 章主要介绍多线程程序可能产生的一些常见的具有隐蔽性的故障以及相应的规避措施。

第 8 章主要介绍在多线程编程中如何更加有效和有效率地使用线程。

第 9 章主要从计算模型的角度出发介绍多线程编程中如何利用线程来提高软件的吞吐率和响应性。

第 10 章主要介绍 Java 平台中多线程程序的调试技巧与测试手段。

第 11 章主要介绍多线程编程的硬件基础以及 Java 平台为屏蔽不同硬件平台的差异而进行的抽象（Java 内存模型）。

第 12 章结合实战案例介绍与 Java 平台中的多线程程序紧密相关的常用性能优化方法与技术。

目标读者

本书适合有一定 Java 语言基础的读者作为入门多线程编程之用，也适合有一定多线程编程经验的读者作为重新梳理知识结构以提升认知层次和参考之用。

本书约定

1. 对于标题中带星号（*）的小节，读者可以选择先浏览一下章节标题就跳过它，之后在阅读后续内容遇到问题时再回头来阅读相应的小节。

2. 斜体格式的方法名表示相应方法为静态方法，例如 `System.currentTimeMillis()`（其中，方法名 `currentTimeMillis` 的字体格式为斜体）。非斜体格式的方法名表示相应方法为相应类的实例方法，例如 `StringBuilder.append(String)` 或者 `StringBuilder.append(String str)` 均表示类 `StringBuilder` 的实例方法 `append`。本书有时候也会省略方法中的形式参数列表。

3. 本书用“/”作为分隔符来表示同一个类的多个方法。例如，`StringBuilder.append(String)/toString()` 表示 `StringBuilder` 的 `append` 方法和 `toString` 方法。

4. 本书所指的 Java 虚拟机（JVM）如无特别说明均特指 Oracle 公司的 HotSpot Java 虚拟机。

5. 就 HotSpot Java 虚拟机而言，JIT（Just In Time）编译器是该 Java 虚拟机的一部分，因此本书有时候并不严格区分 Java 虚拟机和 JIT 编译器。

6. 本书涉及的命令如无特别说明均指 Linux 平台下的命令。

如何阅读本书

本书讲解过程中会涉及一些与多线程编程紧密相关的硬件知识，如果读者对这些知识不太熟悉，可以在阅读过程中参考或者直接先行阅读本书第 11 章前 4 节的内容。

读者也可以先阅读完本书前 4 章的内容，接着就开始集中实践。然后边实践边阅读本书的后续章节，或者在实践过程中遇到问题时再参考本书后续章节的内容。当然，这种阅读方法主要是便于读者尽快上手，并不是说本书后续章节的内容无足轻重。

学习一门新的技术、语言的一个行之有效的方法就是边学习边思考、带着问题在学习过程中寻找答案。因此，本书讲解过程中会穿插一些“扩展阅读”的内容，这些内容多涉及新手在学习多线程编程过程中经常会遇到的一些疑惑和问题。尽管如此，这并不能代替读者自己主动思考并从书中或者其他途径寻找答案。

本书介绍了 Java 标准库中与多线程编程紧密相关的一些类，但是这些内容并不能取代读者亲自阅读 Java 的 API 文档（<http://docs.oracle.com/javase/8/docs/api/>）。

配套源码下载

本书配套源码可以从下列网址下载：

<https://github.com/Viscent/javamtia>

或者，

<http://www.broadview.com.cn/31065>

与作者联系

读者在阅读本书过程中遇到问题或者有任何建议时，可以通过微信公众号 VChannel 与作者联系。

读者服务

轻松注册成为博文视点社区用户 (www.broadview.com.cn), 您即可享受以下服务。

- 下载资源: 本书所提供的示例代码及资源文件均可在【下载资源】处下载。
- 提交勘误: 您对书中内容的修改意见可在【提交勘误】处提交, 若被采纳, 将获赠博文视点社区积分 (在您购买电子书时, 积分可用来抵扣相应金额)。
- 与作者交流: 在页面下方【读者评论】处留下您的疑问或观点, 与作者和其他读者一同学习交流。

页面入口: <http://www.broadview.com.cn/31065>

二维码:



第一部分 多线程编程基础

第 1 章 走近 Java 世界中的线程.....	2
1.1 进程、线程与任务	2
1.2 多线程编程简介	4
1.2.1 什么是多线程编程	4
1.2.2 为什么使用多线程	4
1.3 Java 线程 API 简介	5
1.3.1 线程的创建、启动与运行	5
1.3.2 Runnable 接口	9
1.3.3 线程属性	12
1.3.4 Thread 类的常用方法	14
1.3.5 Thread 类的一些废弃方法	16
1.4 无处不在的线程	17
1.5 线程的层次关系	19
1.6 线程的生命周期状态	21
1.7 线程的监视	22
1.8 多线程编程简单运用实例	26
*1.9 多线程编程的优势和风险	27
1.10 本章小结	29
第 2 章 多线程编程的目标与挑战	31
2.1 串行、并发与并行	31
2.2 竞态	33

2.2.1 二维表分析法：解释竞态的结果.....	37
2.2.2 竞态的模式与竞态产生的条件.....	39
2.3 线程安全性.....	42
2.4 原子性	43
2.5 可见性	49
2.6 有序性	56
2.6.1 重排序的概念	56
2.6.2 指令重排序	57
2.6.3 存储子系统重排序.....	63
2.6.4 貌似串行语义	66
2.6.5 保证内存访问的顺序性.....	68
2.7 上下文切换.....	69
2.7.1 上下文切换及其产生原因.....	69
2.7.2 上下文切换的分类及具体诱因.....	70
2.7.3 上下文切换的开销和测量.....	71
2.8 线程的活性故障.....	73
2.9 资源争用与调度.....	74
2.10 本章小结.....	77
第3章 Java 线程同步机制.....	80
3.1 线程同步机制简介.....	80
3.2 锁概述	81
3.2.1 锁的作用	82
3.2.2 与锁相关的几个概念.....	84
3.2.3 锁的开销及其可能导致的问题.....	86
3.3 内部锁：synchronized 关键字	86
3.4 显式锁：Lock 接口	89
3.4.1 显式锁的调度	91
3.4.2 显式锁与内部锁的比较.....	92
3.4.3 内部锁还是显式锁：锁的选用.....	95
*3.4.4 改进型锁：读写锁	95
3.5 锁的适用场景.....	99
3.6 线程同步机制的底层助手：内存屏障.....	99

*3.7 锁与重排序	102
3.8 轻量级同步机制: volatile 关键字	105
3.8.1 volatile 的作用	105
3.8.2 volatile 变量的开销	111
3.8.3 volatile 的典型应用场景与实战案例	111
3.9 实践: 正确实现看似简单的单例模式	120
3.10 CAS 与原子变量	126
3.10.1 CAS	127
3.10.2 原子操作工具: 原子变量类	129
3.11 对象的发布与逸出	135
3.11.1 对象的初始化安全: 重访 final 与 static	137
3.11.2 安全发布与逸出	142
3.12 本章小结	143
第 4 章 牛刀小试: 玩转线程	148
4.1 挖掘可并发点	148
4.2 新战场上的老武器: 分而治之	148
4.3 基于数据的分割实现并发化	149
4.4 基于任务的分割实现并发化	158
4.4.1 按任务的资源消耗属性分割	159
4.4.2 实战案例的启发	169
4.4.3 按处理步骤分割	171
4.5 合理设置线程数	172
4.5.1 Amdahl's 定律	172
4.5.2 线程数设置的原则	173
4.6 本章小结	177
第 5 章 线程间协作	179
5.1 等待与通知: wait/notify	179
5.1.1 wait/notify 的作用与用法	180
5.1.2 wait/notify 的开销及问题	188
5.1.3 Object.notify()/notifyAll() 的选用	191
*5.1.4 wait/notify 与 Thread.join()	191

5.2	Java 条件变量	192
5.3	倒计时协调器：CountDownLatch	198
5.4	栅栏（CyclicBarrier）	203
5.5	生产者—消费者模式	210
5.5.1	阻塞队列	213
5.5.2	限购：流量控制与信号量（Semaphore）	216
*5.5.3	管道：线程间的直接输出与输入	218
5.5.4	一手交钱，一手交货：双缓冲与 Exchanger	221
5.5.5	一个还是一批：产品的粒度	223
5.5.6	再探线程与任务之间的关系	224
5.6	对不起，打扰一下：线程中断机制	225
5.7	线程停止：看似简单，实则不然	228
5.7.1	生产者—消费者模式中的线程停止	233
5.7.2	实践：Web 应用中的线程停止	233
5.8	本章小结	236
第 6 章	保障线程安全的设计技术	240
*6.1	Java 运行时存储空间	240
6.2	大公无私：无状态对象	243
6.3	以“不变”应万变：不可变对象	248
6.4	我有我地盘：线程特有对象	254
6.4.1	线程特有对象可能导致的问题及其规避	258
6.4.2	线程特有对象的典型应用场景	264
6.5	装饰器模式	265
6.6	并发集合	267
6.7	本章小结	270
第 7 章	线程的活性故障	273
7.1	鹬蚌相争：死锁	273
7.1.1	死锁的检测	274
7.1.2	死锁产生的条件与规避	283
7.1.3	死锁的恢复	296

7.2	沉睡不醒的睡美人：锁死	301
7.2.1	信号丢失锁死	301
7.2.2	嵌套监视器锁死	301
7.3	巧妇难为无米之炊：线程饥饿	307
7.4	屡战屡败，屡败屡战：活锁	307
7.5	本章小结	308
第 8 章	线程管理	310
8.1	线程组	310
8.2	可靠性：线程的未捕获异常与监控	311
8.3	有组织有纪律：线程工厂	316
8.4	线程的暂挂与恢复	318
8.5	线程的高效利用：线程池	320
8.5.1	任务的处理结果、异常处理与取消	326
8.5.2	线程池监控	329
8.5.3	线程池死锁	330
8.5.4	工作者线程的异常终止	330
8.6	本章小结	331
第 9 章	Java 异步编程	333
9.1	同步计算与异步计算	333
9.2	Java Executor 框架	336
9.2.1	实用工具类 Executors	337
9.2.2	异步任务的批量执行：CompletionService	339
9.3	异步计算助手：FutureTask	344
9.3.1	实践：实现 XML 文档的异步解析	345
9.3.2	可重复执行的异步任务	349
9.4	计划任务	352
9.5	本章小结	358

第 10 章	Java 多线程程序的调试与测试	360
10.1	多线程程序的调试技巧	360
10.1.1	使用监视点	360
10.1.2	设置暂挂策略	361
10.2	多线程程序的测试	363
10.2.1	可测试性	364
10.2.2	静态检查工具：FindBugs	369
10.2.3	多线程程序的代码复审	370
10.2.4	多线程程序的单元测试：JCStress	372
10.3	本章小结	375

第二部分 多线程编程进阶

第 11 章	多线程编程的硬件基础与 Java 内存模型	378
11.1	填补处理器与内存之间的鸿沟：高速缓存	378
11.2	数据世界的交通规则：缓存一致性协议	382
11.3	硬件缓冲区：写缓冲器与无效化队列	386
11.3.1	存储转发	388
11.3.2	再探内存重排序	388
11.3.3	再探可见性	391
11.4	基本内存屏障	392
11.5	Java 同步机制与内存屏障	395
11.5.1	volatile 关键字的实现	395
11.5.2	synchronized 关键字的实现	397
11.5.3	Java 虚拟机对内存屏障使用的优化	398
11.5.4	final 关键字的实现	398
11.6	Java 内存模型	399
11.6.1	什么是 Java 内存模型	400
11.6.2	happen(s)-before 关系	401
11.6.3	再探对象的安全发布	407
11.6.4	JSR 133	411
11.7	共享变量与性能	411
11.8	本章小结	411

第 12 章 Java 多线程程序的性能调校	415
12.1 Java 虚拟机对内部锁的优化	415
12.1.1 锁消除	415
12.1.2 锁粗化	417
12.1.3 偏向锁	419
12.1.4 适应性锁	420
12.2 优化对锁的使用	421
12.2.1 锁的开销与锁争用监视	421
12.2.2 使用可参数化锁	424
12.2.3 减小临界区的长度	428
12.2.4 减小锁的粒度	432
12.2.5 考虑锁的替代品	438
12.3 减少系统内耗：上下文切换	438
12.4 多线程编程的“三十六计”：多线程设计模式	440
12.5 性能的隐形杀手：伪共享	441
12.5.1 Java 对象内存布局	442
12.5.2 伪共享的侦测与消除	445
12.6 本章小结	454
Web 参考资源	457
参考文献	463

第一部分

多线程编程基础

- 第 1 章 走近 Java 世界中的线程
 - 第 2 章 多线程编程的目标与挑战
 - 第 3 章 Java 线程同步机制
 - 第 4 章 牛刀小试：玩转线程
 - 第 5 章 线程间协作
 - 第 6 章 保障线程安全的设计技术
 - 第 7 章 线程的活性故障
 - 第 8 章 线程管理
 - 第 9 章 Java 异步编程
 - 第 10 章 Java 多线程程序的调试与测试
-

第 1 章

走近 Java 世界中的线程

To see a world in a grain of sand,

一颗沙里看出一个世界，

And a heaven in a wild flower,

一朵野花里一座天堂，

Hold infinity in the palm of your hand,

把无限放在你的手掌上，

And eternity in an hour.

永恒在一刹那那里收藏。

—— *To see a world in a grain of sand*, William Blake

本章将介绍线程、多线程编程这两个基本概念，以及 Java 平台对线程的实现，并在此基础上给出一个多线程编程的简单应用实例，以便读者对多线程编程有个初步的认识。如果读者想要尽快入门，也可考虑先阅读本章的 1.1 节、1.2 节、1.3 节、1.4 节、1.6 节和 1.8 节，后面遇到疑问时或者有时间时再回头阅读本章的其他小节。这些小节同样也是阅读本书后续章节的基础或者它们能够扩展我们的知识面，有助于我们更好地理解多线程编程。

1.1 进程、线程与任务

进程（Process）是程序的运行实例。例如，一个运行的 Eclipse 就是一个进程。进程与程序之间的关系就好比播放中的视频（如《摩登时代》这部电影）与相应的视频文件（如 mp4 文件）之间的关系，前者从动态的角度刻画事物而后者从静态的角度刻画事物。运行一个 Java 程序的实质是启动一个 Java 虚拟机进程，也就是说一个运行的 Java 程序就是一

个 Java 虚拟机进程¹。例如，运行如清单 1-1 所示的 Java 程序实际上是启动了一个 Java 虚拟机进程。

清单 1-1 一个简单的 Java 程序

```
public class SimpleJavaApp {

    public static void main(String[] args) throws Exception {
        while (true) {
            System.out.println(new Date());
            Thread.sleep(1000);
        }
    }
}
```

运行如清单 1-1 所示的 Java 程序所创建的进程在 Linux 系统下可以使用如下命令查看²：

```
ps -ef | grep "SimpleJavaApp" | grep -v "grep"
```

上述命令的输出类似如下：

```
viscent  8037  6596  0 19:24 pts/6    00:00:00 java io.github.viscent.mtia.ch1.
SimpleJavaApp
```

进程是程序向操作系统申请资源(如内存空间和文件句柄)的基本单位。线程(Thread)是进程中可独立执行的最小单位。例如，一个实现从服务器上下载大文件功能的程序为了提高其文件下载效率可以使用多个线程，这些线程各自独立地从服务器上下载大文件中的一段数据。

一个进程可以包含多个线程。同一个进程中的所有线程共享该进程中的资源，如内存空间、文件句柄等。进程与线程之间的关系，好比一个营业中的饭店与其正在工作的员工之间的关系。一个营业中的饭店对外为顾客提供餐饮服务，而这种服务最终是通过该饭店的员工的工作实现的。这些工作中的员工有的在迎宾，有的在烹调，有的给顾客上菜。他们在其工作过程中共享该饭店的资源，如食材、餐具、清洁用具等。

线程所要完成的计算就被称为任务，特定的线程总是在执行着特定的任务。任务代表线程所要完成的工作，它是一个相对的概念。一个任务可以从服务器上下载一个文件、解压缩一批文件、解压缩一个文件、监视某个文件的最后修改时间等。这些任务也正是相应线程存在的理由。

1 Java Web 应用例外。一个 Java Web 服务器是一个进程，它可以同时运行多个 Java Web 应用。

2 该命令的作用是查找启动命令中包含字符串“SimpleJavaApp”的进程。

1.2 多线程编程简介

1.2.1 什么是多线程编程

函数式编程（Functional Programming）中的函数是基本抽象单位，面向对象编程中的类（Class）是基本抽象单位。类似地，多线程编程就是以线程为基本抽象单位的一种编程范式（Paradigm）。但是，多线程编程又不仅仅是使用多个线程进行编程那么简单，其自身又有其需要解决的问题。当然，多线程编程和面向对象编程是可以相容的，即我们可以在面向对象编程的基础上实现多线程编程，事实上 Java 平台中的一个线程就是一个对象。

多线程编程类似于“和尚挑水”的故事：一个和尚挑水喝，两个和尚担水喝，三个和尚没水喝。在这个故事中，一个和尚挑水会比较吃力，因此每天能运上山的水也非常有限。两个和尚一起担水，每个人都省点儿力，因此他们每天运的水会比一个和尚挑的水要多一些。但是，三个和尚在一起的结果就是大家都不想去打水，最后导致没有水喝！如果把这个故事中的和尚比作线程而把打水比作这些线程所要完成的任务，那么我们不难发现增加线程可能会增加单位时间内完成的任务量，即提高程序的计算效率；但它也可能降低程序的计算效率（如故事中最后大家没有水喝）。可见，多线程编程并非使用多个线程进行编程那么简单。

1.2.2 为什么使用多线程

为什么使用多线程进行编程？弄清楚这个问题有助于我们在实践中做到有的放矢，不至于为了使用多线程而使用多线程。下面我们通过几个多线程编程的典型例子去直观感受一下多线程编程。

某款音乐播放手机软件在其启动的时候会专门启动一个线程用于在用户的手机存储中查找音乐文件，然后自动将这些文件名添加到名为“本地音乐”的播放列表。由于从手机存储器（如 SD 卡）查找特定的文件（音乐文件）是一个相对慢的操作，我们不希望该操作使得该软件启动时显得卡顿。因此，搜索手机本地音乐文件这个任务使用专门的一个线程执行比将其放在负责界面显示的线程（Event Loop 线程）中执行给用户带来的体验要好。

Web 服务器（如 Apache Tomcat）常常在同一时间内会收到多个 HTTP 请求。为了避免一个请求的处理快慢影响到其他请求的处理，绝大多数服务器都会采用一些专门的线程

(工作者线程)负责请求处理,这些线程各自处理分配给它的请求,从而使得一个请求处理的快慢不会对其他请求的处理产生影响(当然,这里的“不影响”是相对的)。这有点像快餐店在点餐顾客多的情况下多开几条点餐线,以减少每个顾客的等待时间。

某系统需要从指定的日志文件中统计出一些信息。而待统计的日志文件中的每个文件可包含上万条记录。若要统计几十个这样的日志文件就会涉及几十万甚至上百万条记录的读取和处理。而读取日志文件所涉及的 I/O 操作又是一个比较慢的操作。因此,这里我们可以使用一个专门的线程负责日志文件的读取。另外,再使用专门的一个线程去负责对读取到内存中的日志记录数据进行统计。这样,使用多线程编程可以使得该统计工具的统计效率尽可能高。

1.3 Java 线程 API 简介

Java 标准库类 `java.lang.Thread` 就是 Java 平台对线程的实现。`Thread` 类或其子类的一个实例就是一个线程。

1.3.1 线程的创建、启动与运行

在 Java 平台中创建一个线程就是创建一个 `Thread` 类(或其子类)的实例。为了讨论的方便,本书后面提到的线程与 `Thread` 实例如无特别说明指的是同一概念。

每个线程都有其要执行的任务。线程的任务处理逻辑可以在 `Thread` 类的 `run` 实例方法中直接实现或者通过该方法进行调用,因此 `run` 方法相当于线程的任务处理逻辑的入口方法,它由 Java 虚拟机在运行相应线程时直接调用,而不是由应用代码进行调用。

运行一个线程实际上就是让 Java 虚拟机执行该线程的 `run` 方法,从而使相应线程的任务处理逻辑代码得以执行。为此,我们首先要启动线程。`Thread` 类的 `start` 方法的作用是启动相应的线程。启动一个线程的实质是请求 Java 虚拟机运行相应的线程,而这个线程具体何时能够运行是由线程调度器(Scheduler)决定的³。因此,`start` 方法调用结束并不意味着相应线程已经开始运行,这个线程可能稍后才被运行,甚至也可能永远不会被运行⁴。

`Thread` 类的两个常用构造器是:`Thread()`和 `Thread(Runnable target)`。相应地,Java 语言中创建线程有两种方式。一种是使用上述第 1 个构造器:定义 `Thread` 类的子类,在该

³ 线程调度器是操作系统的一个部分。

⁴ 详情参见本书的第 7 章内容。

子类中覆盖（Override）run 方法并在该方法中实现线程任务处理逻辑；另一种是使用上述第 2 个构造器：创建一个 java.lang.Runnable 接口的实例，并在该实例的 run 方法中实现任务处理逻辑，然后以该 Runnable 接口实例作为构造器的参数直接创建（new）一个 Thread 类的实例。

在 Java 平台中，每个线程均可以有名字，这个名字便于我们（人）区分不同的线程。

假设我们要创建一个处理任务为打印一行欢迎信息的简单线程。如清单 1-2 和清单 1-3 所示的代码分别展示了用上述两种方法创建相应的线程。

清单 1-2 以定义 Thread 类子类的方式创建线程

```
public class WelcomeApp {

    public static void main(String[] args) {
        // 创建线程
        Thread welcomeThread = new WelcomeThread();

        // 启动线程
        welcomeThread.start();

        // 输出“当前线程”的线程名称
        System.out.printf("1.Welcome! I'm %s.%n", Thread.currentThread().getName());
    }

    // 定义 Thread 类的子类
    class WelcomeThread extends Thread {

        // 在该方法中实现线程的任务处理逻辑
        @Override
        public void run() {
            System.out.printf("2.Welcome! I'm %s.%n", Thread.currentThread().getName());
        }
    }
}
```

在如清单 1-2 所示的代码中，我们定义了类 Thread 类的子类 WelcomeThread，并在该类的 run 方法中实现了任务处理逻辑（打印一行欢迎信息）。然后我们在 main 方法中创建一个 WelcomeThread 的实例 welcomeThread，这就是创建了一个线程（welcomeThread）。接着我们调用 welcomeThread 的 start 方法启动相应的线程。如清单 1-2 所示的程序运行时可能输出如下内容：

```
2.Welcome! I'm Thread-0.
1.Welcome! I'm main.
```

清单 1-3 以创建 Runnable 接口实例的方式创建线程

```
public class WelcomeApp1 {

    public static void main(String[] args) {
        // 创建线程
        Thread welcomeThread = new Thread(new WelcomeTask());

        // 启动线程
        welcomeThread.start();
        // 输出“当前线程”的线程名称
        System.out.printf("1.Welcome! I'm %s.%n", Thread.currentThread().getName());
    }

}

class WelcomeTask implements Runnable {
    // 在该方法中实现线程的任务处理逻辑
    @Override
    public void run() {
        // 输出“当前线程”的线程名称
        System.out.printf("2.Welcome! I'm %s.%n", Thread.currentThread().getName());
    }

}
```

在如清单 1-3 所示的代码中,我们先定义了一个 Runnable 接口的实现类 WelcomeTask,并在该类的 run 方法中实现了我们要创建的线程的任务处理逻辑(打印一行欢迎信息)。接着,我们在 main 方法中创建一个 WelcomeTask 类的实例并以该实例作为构造器参数直接通过 new 创建一个 Thread 实例,这就创建了一个线程(welcomeThread)。如清单 1-3 所示的程序运行时可能输出如下内容:

```
2.Welcome! I'm Thread-0.
1.Welcome! I'm main.
```

多次运行如清单 1-2 或者清单 1-3 所示的程序,我们可以发现这两个程序的输出也可能是:

```
1.Welcome! I'm main.
2.Welcome! I'm Thread-0.
```

这是因为,打印“2”开头消息的线程(welcomeThread)的启动虽然是在打印上面的“1”开头消息的语句之前,但是这并不意味着 welcomeThread 线程会在打印“1”开头消

息的语句被执行之前得以运行。

不管是采用哪种方式创建线程，一旦线程的 `run` 方法执行（由 Java 虚拟机调用）结束，相应的线程的运行也就结束了。当然，`run` 方法执行结束包括正常结束（`run` 方法返回）以及代码中抛出异常而导致的中止。运行结束的线程所占用的资源（如内存空间）会如同其他 Java 对象一样被 Java 虚拟机垃圾回收。

线程属于“一次性用品”，我们不能通过重新调用一个已经运行结束的线程的 `start` 方法来使其重新运行。事实上，`start` 方法也只能被调用一次，多次调用同一个 `Thread` 实例的 `start` 方法会导致其抛出 `IllegalThreadStateException` 异常。

在 Java 平台中，一个线程就是一个对象，对象的创建离不开内存空间的分配。创建一个线程与创建其他类型的 Java 对象所不同的是，Java 虚拟机会为每个线程分配调用栈（Call Stack）所需的内存空间。调用栈用于跟踪 Java 代码（方法）间的调用关系以及 Java 代码对本地代码（Native Code，通常是 C 代码）的调用。另外，Java 平台中的每个线程可能还有一个内核线程（具体与 Java 虚拟机的实现有关）与之对应⁵。因此相对来说，创建线程对象比创建其他类型的对象的成本要高一些。

Java 平台中的任意一段代码（比如一个方法）总是由确定的线程负责执行的，这个线程就相应地被称为这段代码的执行线程。同一段代码可以被多个线程执行。例如，如清单 1-3 所示的代码中创建的 `WelcomeTask` 实例的 `run` 方法是运行在 `welcomeThread` 这个线程中的。如果我们以同一个 `WelcomeTask` 实例作为构造器参数创建多个 `WelcomeThread` 实例（线程）的话，那么这个 `WelcomeTask` 实例的 `run` 方法就会被多个线程执行。任意一段代码都可以通过调用 `Thread.currentThread()` 来获取这段代码的执行线程，这个线程就被称为当前线程。由于同一段代码可以被多个线程执行，因此当前线程是相对的，即概念层次上的当前线程（即 `Thread.currentThread()` 的返回值）在代码实际运行的时候可能对应着不同的线程（对象）。这就好比大家都自称“本人”（当前线程），“本人”这个词由张三来说就是指张三（线程 X），而由李四来说则指李四（线程 Y）。

我们知道线程的 `run` 方法总是由 Java 虚拟机直接调用的。尽管如此，Java 语言并不阻止我们直接调用 `run` 方法，这是因为：首先，线程在 Java 平台中也是一个对象，其次毕竟 `run` 方法也是一个 `public` 方法。但是，多数情况下我们不能这样做，因为这样做有违创建线程（对象）的初衷。例如，运行如清单 1-4 所示的程序（改自清单 1-3），我们可以看到该程序的输出类似如下：

5 由操作系统内核直接进行管理和调度的线程，它与具体的操作系统平台有关。

```

2.Welcome! I'm main.
1.Welcome! I'm main.
2.Welcome! I'm Thread-0.

```

可见，线程 `welcomeThread` 的 `run` 方法被执行了两次：一次是 Java 虚拟机直接调用执行，此时 `welcomeThread` 的 `run` 方法是运行在自己的线程中的（从打印消息中的线程名称 `Thread-0` 可以看出这点）；另一次是由我们的应用代码直接调用执行，此时 `welcomeThread` 的 `run` 方法实际上运行在 `main` 线程中（从打印消息中的线程名称 `main` 可以看出这一点）。因此，如果我们没有启动线程而是在应用代码中直接调用线程的 `run` 方法的话，那么这个线程的 `run` 方法其实运行在当前线程（即 `run` 方法的调用方代码的执行线程）之中而不是运行在其自身线程之中，从而违背了创建线程的初衷。

清单 1-4 应用代码直接调用线程的 `run` 方法（避免这样做!）

```

public class WelcomeApp2 {
    public static void main(String[] args) {

        // 创建线程
        Thread welcomeThread = new Thread(new Runnable() {
            @Override
            public void run() {
                System.out.printf("2.Welcome! I'm %s.%n", Thread.currentThread()
                    .getName());
            }
        });

        // 启动线程
        welcomeThread.start();
        // 这里直接调用线程的 run 方法，仅是出于演示的目的
        welcomeThread.run();
        System.out.printf("1.Welcome! I'm %s.%n", Thread.currentThread().getName());
    }
}

```

1.3.2 Runnable 接口

`Runnable` 接口只定义了一个方法，该方法的声明如下：

```
public void run()
```

`Runnable` 接口可以被看作对任务进行的抽象，任务的处理逻辑就体现在 `run` 方法之中。`Thread` 类实际上是 `Runnable` 接口的一个实现类，其对 `Runnable` 接口的实现如图 1-1 所示。


```
public void run() {  
    if (target != null) {  
        target.run();  
    }  
}
```

图 1-1 Thread.run()源码

可见，Thread 类的 run 方法中实现的逻辑是如果 target 不为 null，那么就调用 target.run()，否则它什么也不做。其中，实例变量 target 的类型为 Runnable。如果相应的线程实例是通过构造器 Thread(Runnable target) 创建的，那么 target 的值为构造器中的参数值，否则 target 的值为 null。因此，Thread 类所实现的任务处理逻辑是要么什么也不做（target 为 null），要么直接执行 target 所引用的 Runnable 实例所实现的任务处理逻辑。Thread 类的 run 方法的这种处理逻辑决定了创建线程的两种方式：一种是在 Thread 子类的 run 方法中直接实现任务处理逻辑，另一种是在一个 Runnable 实例中实现任务处理逻辑，该逻辑由 Thread 类的 run 方法负责调用。

扩展阅读 线程两种创建方式的区别

从面向对象编程的角度来看：第 1 种创建方式（创建 Thread 类的子类）是一种基于继承（Inheritance）的技术，第 2 种创建方式（以 Runnable 接口实例为构造器参数直接通过 new 创建 Thread 实例）是一种基于组合（Composition）的技术。由于组合相对继承来说，其类和类之间的耦合性（Coupling）更低，因此它也更加灵活。一般我们认为组合是优先选用的技术。

从对象共享的角度来看：第 2 种创建方式意味着多个线程实例可以共享同一个 Runnable 实例。在某些情况下这可能导致程序的运行结果出乎我们的意料。例如，在如清单 1-5 所示的程序中，我们分别以第 2 种和第 1 种方式各自创建了若干线程。假设这个程序运行在处理器个数为 4 的主机上，那么该程序的输出中的“CountingTask:”后面跟的数字最大的数也可能仍小于 800（ $=2 \times 4 \times 100$ ），而“CountingThread:”后面跟的数字始终都是 100。对于这个结果的解释会涉及我们将在第 2 章中提到的竞态和线程安全这两个概念。

从对象创建成本的角度来看：Java 中的线程实例是一个“特殊”的 Runnable 实例，因为在创建它的时候 Java 虚拟机会为其分配调用栈空间、内核线程等资源。因此，创建一个线程实例比起创建一个普通的 Runnable 实例来说，其成本要相对昂贵一点。所以，如果创建 Runnable 实例再将其作为方法参数传递给其他对象使用（JDK 标准库中有不少 API 都使用了 Runnable 接口）而不必利用它来创建相应的线程（即第 2 种线程创建方式）即可满足我们的计算需要，那么就不要再创建线程实例。

清单 1-5 线程的两种创建方式的区别

```
public class ThreadCreationCmp {

    public static void main(String[] args) {
        Thread t;
        CountingTask ct = new CountingTask();

        // 获取处理器个数
        final int numberOfProceesors = Runtime.getRuntime().availableProcessors();

        for (int i = 0; i < 2 * numberOfProceesors; i++) {

            // 直接创建线程
            t = new Thread(ct);
            t.start();
        }

        for (int i = 0; i < 2 * numberOfProceesors; i++) {

            // 以子类的方式创建线程
            t = new CountingThread();

            t.start();
        }
    }

    static class Counter {
        private int count = 0;

        public void increment() {
            count++;
        }

        public int value() {
            return count;
        }
    }

    static class CountingTask implements Runnable {
        private Counter counter = new Counter();

        @Override
        public void run() {
            for (int i = 0; i < 100; i++) {
                doSomething();
                counter.increment();
            }
        }
    }
}
```

```
    }  
    System.out.println("CountingTask:" + counter.value());  
}  
  
private void doSomething() {  
    // 使当前线程休眠随机时间  
    Tools.randomPause(80);  
}  
  
}  
  
static class CountingThread extends Thread {  
    private Counter counter = new Counter();  
  
    @Override  
    public void run() {  
        for (int i = 0; i < 100; i++) {  
            doSomething();  
            counter.increment();  
        }  
        System.out.println("CountingThread:" + counter.value());  
    }  
  
    private void doSomething() {  
        // 使当前线程休眠随机时间  
        Tools.randomPause(80);  
    }  
  
}  
}
```

1.3.3 线程属性

线程的属性包括线程的编号（ID）、名称（Name）、线程类别（Daemon）和优先级（Priority），详情如表 1-1 所示。

表 1-1 线程的属性

属 性	属性类型及用途	只读属性	重要注意事项
编号 (ID)	类型: long。用于标识不同的线程。不同的线程拥有不同的编号	是	某个编号的线程运行结束后, 该编号可能被后续创建的线程使用。不同线程拥有的编号虽然不同, 但是这种编号的唯一性只在 Java 虚拟机的一次运行有效。也就是说重启一个 Java 虚拟机 (如重启 Web 服务器) 后, 某些线程的编号可能与上次 Java 虚拟机运行的某个线程的编号一样, 因此该属性的值不适合用作某种唯一标识, 特别是作为数据库中的唯一标识 (如主键)
名称 (Name)	类型: String。面向人 (而非机器) 的一个属性, 用于区分不同的线程。默认值与线程的编号有关, 默认值的格式为: “Thread-线程编号”, 如 “Thread-0”	否	Java 并不禁止我们将不同的线程的名称属性设置为相同的值。尽管如此, 设置线程的名称属性有助于代码调试和问题定位
线程类别 (Daemon)	类型: boolean。值为 true 表示相应的线程为守护线程, 否则表示相应的线程为用户线程。该属性的默认值与相应线程的父线程的该属性的值相同 ⁶	否	该属性必须在相应线程启动之前设置, 即对 setDaemon 方法的调用必须在对 start 方法的调用之前, 否则 setDaemon 方法会抛出 IllegalStateException 异常。负责一些关键任务处理的线程不适宜设置为守护线程
优先级 (Priority)	类型: int。该属性本质上是给线程调度器的提示, 用于表示应用程序希望哪个线程能够优先得以运行。Java 定义了 1~10 的 10 个优先级。默认值一般为 5 (表示普通优先级)。对于具体的一个线程而言, 其优先级的默认值与其父线程 (创建该线程的线程) 的优先级值相等	否	一般使用默认优先级即可。不恰当地设置该属性值可能导致严重的问题 (线程饥饿)

⁶ 守护线程和父线程的概念下文会介绍。

通过名称属性，我们可以为每个线程设置一个便于区分不同线程的名称。虽然 Java 虚拟机并不要求每个线程的名称都不同，但是设置该属性有助于程序调试和问题定位。因此，我们建议为每个线程都设置一个简短而又能够体现其作用或其实现的功能的名称。

线程的属性除了编号外，其他属性都是可读写的属性，即 Thread 类提供了相应的 get 方法和 set 方法用于读取或者设置相应的属性。例如，getName 方法可返回线程的名称属性值而 setName 方法则可以设置线程的名称属性值。

Java 线程的优先级属性本质上只是一个给线程调度器的提示信息，以便于线程调度器决定优先调度哪些线程运行。它并不能保证线程按照其优先级高低的顺序运行。注意，Java 线程的优先级使用不当或者滥用则可能导致某些线程永远无法得到运行，即产生了线程饥饿（Thread Starvation）。因此，线程的优先级并不是设置得越高越好；一般情况下使用普通优先级即可，即不必设置线程的优先级属性。

按照线程是否会阻止 Java 虚拟机正常停止，我们可以将 Java 中的线程分为守护线程（Daemon Thread）和用户线程（User Thread，也称非守护线程）⁷。线程的 daemon 属性用于表示相应线程是否为守护线程。用户线程会阻止 Java 虚拟机的正常停止，即一个 Java 虚拟机只有在其所有用户线程都运行结束（即 Thread.run()调用未结束）的情况下才能正常停止。而守护线程则不会影响 Java 虚拟机的正常停止，即应用程序中有守护线程在运行也不影响 Java 虚拟机的正常停止。因此，守护线程通常用于执行一些重要性不是很高的任务，例如用于监视其他线程的运行情况。

如果 Java 虚拟机是被强制停止的，比如在 Linux 系统下使用 kill 命令强制终止一个 Java 虚拟机进程⁸，那么即使是用户线程也无法阻止 Java 虚拟机的停止。

1.3.4 Thread 类的常用方法

Thread 类的常用方法如表 1-2 所示。这里列出这些方法只是为了便于读者快速了解 Thread 类的相关 API，它并不能取代读者在阅读本书和工作过程中阅读 Java 标准库文档（JavaDoc）⁹。

7 Java 虚拟机正常停止指不是通过 System.exit 调用也不是通过强制终止进程（如在 Linux 系统下使用 kill 命令停止 Java 进程）实现的 Java 虚拟机停止。

8 具体命令为：kill -9 PID，其中参数 PID 为 Java 进程的进程 ID。

9 参见：<http://docs.oracle.com/javase/7/docs/api/>。

表 1-2 Thread 类的常用方法

方 法	功 能	备 注
static Thread currentThread()	返回当前线程，即当前代码的 执行线程（对象）	同一段代码对 Thread.currentThread() 的调用，其返回值可能对应着不同的线 程（对象）
void run()	用于实现线程的任务处理逻辑	该方法是由 Java 虚拟机直接调用的， 一般情况下应用程序不应该调用该方法
void start()	启动相应线程	该方法的返回并不代表相应的线程已 经被启动。一个 Thread 实例的 start 方法 只能够被调用一次，多次调用会导致异 常的抛出
void join()	等待相应线程运行结束	若线程 A 调用线程 B 的 join 方法，那 么线程 A 的运行会被暂停，直到线程 B 运行结束
static void yield()	使当前线程主动放弃其对处理 器的占用，这可能导致当前线程 被暂停	这个方法是不可靠的。该方法被调用 时当前线程可能仍然继续运行（视系统 当前的运行状况而定）
static void sleep(long millis)	使当前线程休眠（暂停运行） 指定的时间	

Java 中的任何一段代码总是执行在某个线程之中。执行当前代码的线程就被称为当前线程，Thread.currentThread()可以返回当前线程。由于同一段代码可能被不同的线程执行，因此当前线程是相对的，即 Thread.currentThread()的返回值在代码实际运行的时候可能对应着不同的线程（对象）。

join 方法的作用相当于执行该方法的线程和线程调度器说：“我得先暂停一下，等到另外一个线程运行结束后我才能继续（干活）。”我们会在第 5 章中进一步讲解该方法。

yield 静态方法的作用相当于执行该方法的线程对线程调度器说：“我现在不急，如果别人需要处理器资源的话先给他用吧。当然，如果没有其他人要用，我也不介意继续占用。”

sleep 静态方法的作用相当于执行该方法的线程对线程调度器说：“我想小憩一会儿，过段时间再叫醒我继续干活吧。”使用 sleep 静态方法可以实现一个简易的倒计时器，如清单 1-6 所示。

清单 1-6 简易的倒计时器

```

public class SimpleTimer {
    private static int count;

    public static void main(String[] args) {
        count = args.length >= 1 ? Integer.valueOf(args[0]) : 60;
        int remaining;
        while (true) {
            remaining = countDown();
            if (0 == remaining) {
                break;
            } else {
                System.out.println("Remaining " + count + " second(s)");
            }

            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                // 什么也不做
            }
        }
        System.out.println("Done.");
    }

    private static int countDown() {
        return count--;
    }
}

```

1.3.5 Thread 类的一些废弃方法

由于 Java 虚拟机实现得有些问题,因此 Thread 类的有些方法已经被废弃(Deprecated)了¹⁰,在新写的代码中应该避免使用这些方法。部分废弃的方法如表 1-3 所示。

表 1-3 Thread 类部分废弃的方法

方 法	功 能
stop	停止线程的运行
suspend	暂停线程的运行
resume	使被暂停的线程继续运行

¹⁰ 具体原因参见: <http://docs.oracle.com/javase/7/docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html>。

虽然表 1-3 的有些方法并没有相应的替代品，即实现同样功能的其他方法；但是，停止线程以及暂停和继续运行线程这些功能我们可以使用其他办法来实现。详情请参见本书的第 8 章。

1.4 无处不在的线程

Java 平台本身就是一个多线程的平台。除了 Java 开发人员自己创建和使用的线程（即 Thread 类或其子类的实例），Java 平台中其他由 Java 虚拟机创建、使用的线程也随处可见。当然，这些线程也是各自有其处理任务。

Java 虚拟机启动的时候会创建一个 main 线程，该线程负责执行 Java 程序的入口方法（main 方法），如清单 1-7 所示。

清单 1-7 Java 代码的执行线程

```
public class JavaThreadAnywhere {

    public static void main(String[] args) {
        // 获取当前线程
        Thread currentThread = Thread.currentThread();

        // 获取当前线程的线程名称
        String currentThreadName = currentThread.getName();

        System.out.printf("The main method was executed by thread:%s",
            currentThreadName);
        Helper helper = new Helper("Java Thread AnyWhere");
        helper.run();
    }

    static class Helper implements Runnable {
        private final String message;

        public Helper(String message) {
            this.message = message;
        }

        private void doSomething(String message) {
            // 获取当前线程
            Thread currentThread = Thread.currentThread();

            // 获取当前线程的线程名称
            String currentThreadName = currentThread.getName();
```



```
        System.out.printf("The doSomething method was executed by thread:%s",
            currentThreadName);
        System.out.println("Do something with " + message);
    }

    @Override
    public void run() {
        doSomething(message);
    }
}
```

清单 1-7 中的程序利用 `currentThread` 方法打印出当前线程的线程名，其执行时输出如下：

```
The main method was executed by thread:main
The doSomething method was executed by thread:main
Do something with Java Thread AnyWhere
```

从上面的输出可以看出，类 `JavaThreadAnywhere` 的 `main` 方法以及类 `Helper` 的 `doSomething` 方法均由 `main` 线程负责执行。这里，`main` 线程的处理任务就是清单 1-7 中的 `main` 方法所实现的处理逻辑，包括打印当前线程的线程名称等。

Web 应用中的 `Servlet` 类的 `doGet`、`doPost` 等方法也总是由确定的线程负责执行的（具体与所使用的 Web 容器有关），如清单 1-8 所示。

清单 1-8 Servlet 类的执行线程

```
@WebServlet("/echo")
public class EchoServlet extends HttpServlet {
    private static final long serialVersionUID = 4787580353870831328L;

    @Override
    protected void
        doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        // 获取当前线程
        Thread currentThread = Thread.currentThread();
        // 获取当前线程的线程名称
        String currentThreadName = currentThread.getName();
        response.setContentType("text/plain");
        try (PrintWriter pwr = response.getWriter()) {
            // 输出处理当前请求的线程的名称
            pwr.printf("This request was handled by thread:%s\n", currentThreadName);
        }
    }
}
```

```

        pwr.flush();
    }
}
}

```

如清单 1-8 所示的 Servlet 类在处理一个 HTTP GET 请求时,其输出可能如下所示(以 Tomcat 作为 Web 容器):

```
This request was handled by thread:http-8080-2
```

这说明此时 EchoServlet 的 doGet 方法在名为“http-8080-2”的线程中执行。因此我们也可以说,此时上述名为“http-8080-2”的线程的处理任务就是执行 EchoServlet 的 doGet 方法所实现的 HTTP 请求处理逻辑。

在多线程编程中,弄清楚一段代码具体是由哪个(或者哪种)线程去负责执行的这点很重要,这关系到性能、线程安全等问题¹¹。本书的后续章节会体现这点。

Java 虚拟机垃圾回收器(Garbage Collector)负责对 Java 程序中不再使用的内存空间进行回收,而这个回收的动作实际上也是通过专门的线程(垃圾回收线程)实现的,这些线程由 Java 虚拟机自行创建。从垃圾回收的角度看,Java 平台中的线程可以分为垃圾回收线程和应用线程。应用线程由 Java 应用程序开发者创建。例如,如清单 1-2 所示的程序所创建的线程就是应用线程。

为了提高 Java 代码的执行效率,Java 虚拟机中的 JIT(Just In Time)编译器会动态地将 Java 字节码(Byte Code)编译为 Java 虚拟机宿主机处理器可直接执行的机器码(本地代码)。这个动态编译的过程实际上是由 Java 虚拟机创建的专门的线程负责执行的。

Java 平台中的线程随处可见,这些线程各自都有其处理任务。

1.5 线程的层次关系

Java 平台中的线程不是孤立的,线程与线程之间总是存在一些联系。假设线程 A 所执行的代码创建了线程 B,那么,习惯上我们称线程 B 为线程 A 的子线程,相应地线程 A 就被称为线程 B 的父线程。例如,清单 1-2 中的线程 welcomeThread 是 main 线程的子线程,main 线程是该线程的父线程。子线程所执行的代码还可以创建其他线程,因此一个子线程也可以是其他线程的父线程。所以,父线程、子线程是一个相对的称呼。

¹¹ 本书第 2 章会介绍线程安全这个概念。

线程间的这种父子关系就被称为线程的层次关系。由于 Java 虚拟机创建的 `main` 线程（也被称为主线程）负责执行 Java 程序的入口方法 `main` 方法，因此 `main` 方法中直接创建的线程都是 `main` 线程的子线程。这些子线程所执行的代码又可能创建其他线程。因此，这就形成了 Java 程序的线程层次关系，如图 1-2 所示。

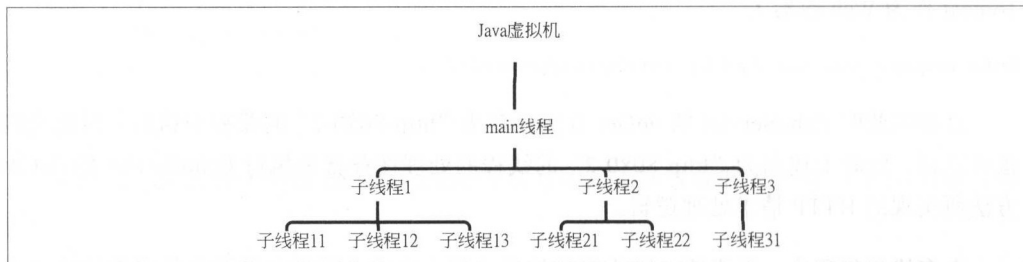


图 1-2 Java 程序的线程层次关系

理解线程的层次关系有助于我们理解 Java 应用程序的结构，也有助于我们后续阐述其他概念。

在 Java 平台中，一个线程是否是一个守护线程默认取决于其父线程：默认情况下父线程是守护线程，则子线程也是守护线程；父线程是用户线程，则子线程也是用户线程。另外，父线程在创建子线程后启动子线程之前可以调用该线程的 `setDaemon` 方法，将相应的线程设置为守护线程（或者用户线程）。

一个线程的优先级默认值为该线程的父线程的优先级，即如果我们没有设置或者更改一个线程的优先级，那么这个线程的优先级的值与父线程的优先级的值相等。

不过，Java 平台中并没有 API 用于获取一个线程的父线程，或者获取一个线程的所有子线程。并且，父线程和子线程之间的生命周期也没有必然的联系。比如父线程运行结束后，子线程可以继续运行，子线程运行结束也不妨碍其父线程继续运行。

习惯上，我们也称某些子线程为工作者线程（Worker Thread）或者后台线程（Background Thread）。工作者线程通常是其父线程创建来用于专门负责某项特定任务的执行的。例如，清单 1-8 中执行 `doGet` 方法对请求进行处理的线程通常被称为 Web 服务器的工作者线程。Java 虚拟机中对内存进行回收的线程通常被称为 GC（Garbage Collection）工作者线程。

1.6 线程的生命周期状态

在 Java 语言中，一个线程从其创建、启动到其运行结束的整个生命周期可能经历若干状态，如图 1-3 所示。

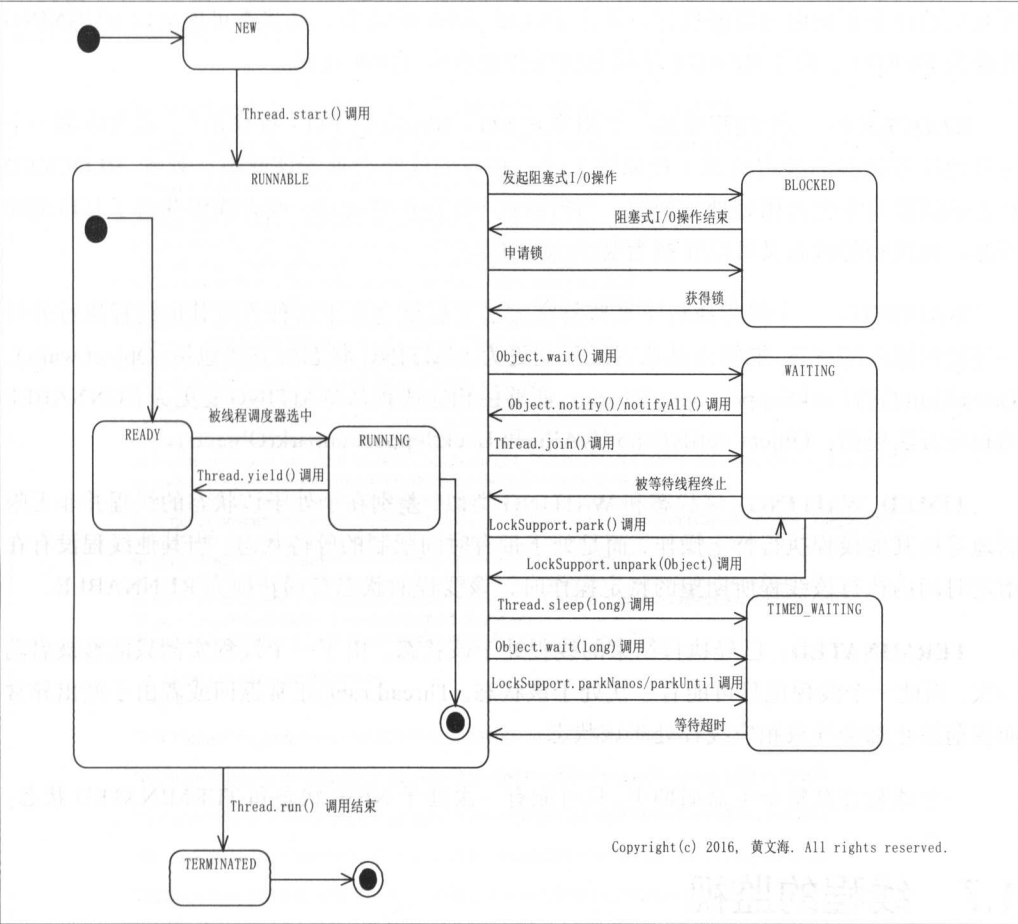


图 1-3 Java 线程的状态

Java 线程的状态可以使用监控工具查看，也可以通过 `Thread.getState()` 调用来获取。`Thread.getState()` 的返回值类型 `Thread.State` 是一个枚举类型 (Enum)。`Thread.State` 所定义的线程状态包括以下几种。

NEW：一个已创建而未启动的线程处于该状态。由于一个线程实例只能够被启动一

次，因此一个线程只可能有一次处于该状态。

RUNNABLE：该状态可以被看成一个复合状态。它包括两个子状态：READY 和 RUNNING。前者表示处于该状态的线程可以被线程调度器（Scheduler）进行调度而使之处于 RUNNING 状态。后者表示处于该状态的线程正在运行，即相应线程对象的 run 方法所对应的指令正在由处理器执行。执行 Thread.yield() 的线程，其状态可能会由 RUNNING 转换为 READY。处于 READY 子状态的线程也被称为活跃线程。

BLOCKED：一个线程发起一个阻塞式 I/O（Blocking I/O）操作后¹²，或者申请一个由其他线程持有的独占资源（比如锁）时，相应的线程会处于该状态。处于 BLOCKED 状态的线程并不会占用处理器资源。当阻塞式 I/O 操作完成后，或者线程获得了其申请的资源，该线程的状态又可以转换为 RUNNABLE。

WAITING：一个线程执行了某些特定方法之后就会处于这种等待其他线程执行另外一些特定操作的状态。能够使其执行线程变更为 WAITING 状态的方法包括：Object.wait()、Thread.join() 和 LockSupport.park(Object)。能够使相应线程从 WAITING 变更为 RUNNABLE 的相应方法包括：Object.notify()/notifyAll() 和 LockSupport.unpark(Object)。

TIMED_WAITING：该状态和 WAITING 类似，差别在于处于该状态的线程并非无限制地等待其他线程执行特定操作，而是处于带有时间限制的等待状态。当其他线程没有在指定时间内执行该线程所期望的特定操作时，该线程的状态自动转换为 RUNNABLE。

TERMINATED：已经执行结束的线程处于该状态。由于一个线程实例只能够被启动一次，因此一个线程也只可能有一次处于该状态。Thread.run() 正常返回或者由于抛出异常而提前终止都会导致相应线程处于该状态。

一个线程在其整个生命周期中，只可能有一次处于 NEW 状态和 TERMINATED 状态。

1.7 线程的监视

一个真实的 Java 系统运行时往往有上百个线程在运行，如果没有相应的工具能够对这些线程进行监视，那么这些线程对于我们来说就成了黑盒。而我们在开发过程中进行代码调试、定位问题甚至是定位线上环境（生产环境）中的问题时往往都需要将线程变为白盒，即我们要能够知道系统中特定时刻存在哪些线程、这些线程处于什么状态以及这些线程具体是在做什么事情这些信息。

¹² 如文件读写和阻塞式 Socket 读写。

对线程进行监视的主要途径是获取并查看程序的线程转储 (Thread Dump)。一个程序的线程转储包含了获取这个线程转储的那一刻该程序的线程信息。这些信息包括程序中有哪些线程以及这些线程的具体信息。Java 程序的线程转储 (如图 1-4 所示) 包含的线程具体信息包括线程的属性 (ID、名称、优先级等)、生命周期状态、线程的调用栈 (Call Stack) 以及锁 (第 3 章会介绍这个概念) 的相关信息等。通过查看调用栈我们就能够了解线程的执行情况 (具体在干什么)。

```
Full thread dump Java HotSpot(TM) 64-Bit Server VM (25.40-b25 mixed mode):

"main" #1 prio=5 os_prio=0 tid=0x00007f5d2400b800 nid=0x229b runnable [0x00007f5d2a52d000]
  java.lang.Thread.State: RUNNABLE
  at java.io.FileInputStream.readBytes(Native Method)
  at java.io.FileInputStream.read(FileInputStream.java:255)
  at java.io.BufferedInputStream.read1(BufferedInputStream.java:284)
  at java.io.BufferedInputStream.read(BufferedInputStream.java:345)
  - locked <0x00000000fe01d6e0> (a java.io.BufferedInputStream)
  at sun.nio.cs.StreamDecoder.readBytes(StreamDecoder.java:284)
  at sun.nio.cs.StreamDecoder.implRead(StreamDecoder.java:326)
  at sun.nio.cs.StreamDecoder.read(StreamDecoder.java:178)
  - locked <0x00000000fe1bf600> (a java.io.InputStreamReader)
  at java.io.InputStreamReader.read(InputStreamReader.java:184)
  at java.io.BufferedReader.fill(BufferedReader.java:161)
  at java.io.BufferedReader.readLine(BufferedReader.java:324)
  - locked <0x00000000fe1bf600> (a java.io.InputStreamReader)
  at java.io.BufferedReader.readLine(BufferedReader.java:389)
  at io.github.viscent.mtia.ch4.case02.CaseRunner4_2$.hasMoreElements(CaseRunner4_2.java:102)
  at java.io.SequenceInputStream.nextStream(SequenceInputStream.java:109)
  at java.io.SequenceInputStream.<init>(SequenceInputStream.java:69)
  at io.github.viscent.mtia.ch4.case02.CaseRunner4_2.createInputStream(CaseRunner4_2.java:92)
  at io.github.viscent.mtia.ch4.case02.CaseRunner4_2.main0(CaseRunner4_2.java:39)
  at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
  at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
  at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
  at java.lang.reflect.Method.invoke(Method.java:497)
  at io.github.viscent.mtia.util.AppWrapper.main(AppWrapper.java:77)

  Locked ownable synchronizers:
  - None

"VM Thread" os_prio=0 tid=0x00007f5d24079000 nid=0x22a3 runnable

"GC task thread#0 (ParallelGC)" os_prio=0 tid=0x00007f5d24020800 nid=0x229f runnable

"GC task thread#1 (ParallelGC)" os_prio=0 tid=0x00007f5d24022800 nid=0x22a0 runnable

"GC task thread#2 (ParallelGC)" os_prio=0 tid=0x00007f5d24024000 nid=0x22a1 runnable

"GC task thread#3 (ParallelGC)" os_prio=0 tid=0x00007f5d24026000 nid=0x22a2 runnable

"VM Periodic Task Thread" os_prio=0 tid=0x00007f5d240c6000 nid=0x22ab waiting on condition

JNI global references: 6
```

图 1-4 线程转储样品 (省略部分内容)

获取线程转储的方法如表 1-4 所示。

表 1-4 获取线程转储的方法

平 台	获取途径		备 注
平台无关	方法 1	执行命令：jstack -l PID ^{①,②}	① PID 为 Java 程序的进程 ID。Java 程序的进程 ID 可以使用 JDK 的 jps 命令（可执行文件是 JDK 主目录/bin/jps）或者 Linux 的 ps 命令来获取 ② jstack 是 Oracle JDK 自带的一个工具，其可执行文件是：JDK 主目录/bin/jstack；Windows 版 JDK 自 JDK 1.6 开始提供该工具；参见： http://docs.oracle.com/javase/7/docs/technotes/tools/share/jstack.html ③ jvisualvm 是 Oracle JDK 自带的一个工具，其可执行文件是：JDK 主目录/bin/jvisualvm。参见图 1-5 ④ JMC 是 Oracle JDK 1.8 开始自带的一个工具，其可执行文件是：JDK 主目录/bin/jmc。JMC 支持 Eclipse 插件。参见图 1-6 ⑤ 参见： https://access.redhat.com/solutions/19170
	方法 2	单击图形化工具 jvisualvm 中的 Thread Dump 按钮 ^③	
	方法 3	使用图形化工具 Java Mission Control（JMC） ^④	
特定于 Linux	方法 4	执行命令：kill -3 PID	
	方法 5	在启动 Java 程序的控制台按下“CTRL+”组合键	
特定于 Windows	方法 6	在启动 Java 程序的命令行提示窗口中按下“CTRL+Break”组合键 ^⑤	

JDK 自带的工具 jvisualvm 适合于在开发和测试环境下监视 Java 系统中的线程情况。jvisualvm 不仅可以用来获取线程转储，它还支持直接选中一个线程来查看该线程的调用栈¹³。图 1-5 展示了使用 jvisualvm 监视一个运行的 Eclipse 实例中的线程情况。

Java Mission Control（JMC）不仅能够用来获取与查看线程转储，它还支持其他功能，如图 1-6 所示。

13 jvisualvm 需要安装 Threads Inspector 插件（Plugin）才能够查看选中线程的调用栈。

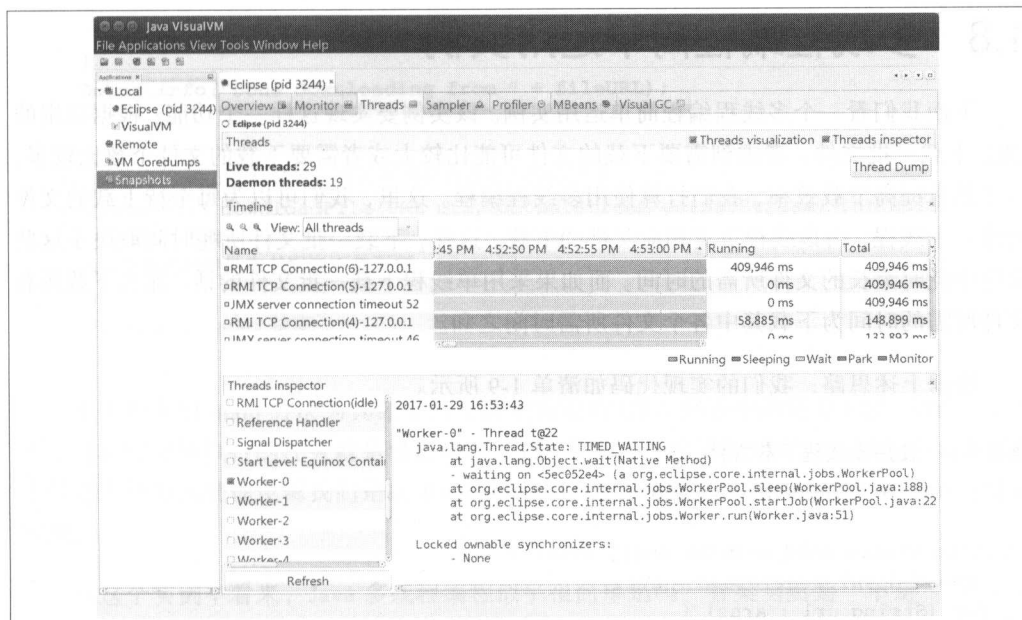


图 1-5 使用 jvisualvm 监视 Java 线程

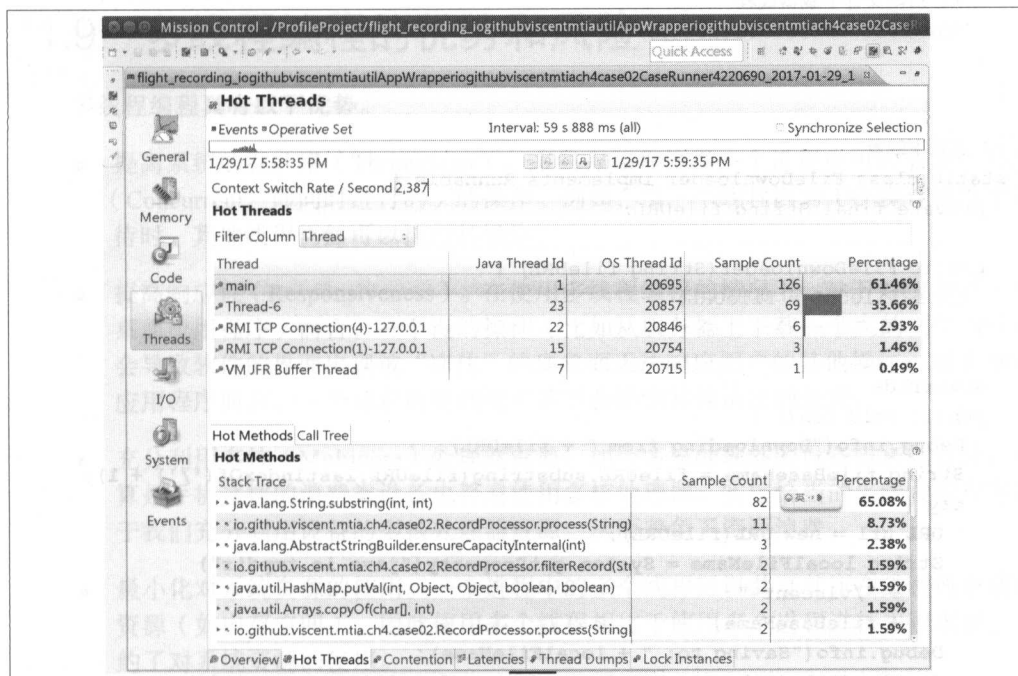


图 1-6 使用 JMC 监视 Java 线程

1.8 多线程编程简单运用实例

下面我们看一个多线程编程简单运用实例。该实例要实现这样一个功能：根据指定的 URL 下载一批文件。考虑到需要下载的文件可能比较大或者需要下载的文件个数比较多，为了提高下载效率，我们打算使用多线程编程。这里，我们可以为每个待下载的文件创建一个线程，由该线程负责相应文件的下载。这样，下载一批文件所耗时间取决于这些文件中耗时最长的文件所需的时间。而如果采用单线程下载一批文件的话，那么下载所有文件所需的时间为下载其中各个文件所需时间之和。

根据上述思路，我们的实现代码如清单 1-9 所示。

清单 1-9 使用多线程下载文件

```
public class FileDownloaderApp {

    public static void main(String[] args) {
        Thread downloaderThread = null;
        for (String url : args) {
            // 创建文件下载器线程
            downloaderThread = new Thread(new FileDownloader(url));
            // 启动文件下载器线程
            downloaderThread.start();
        }
    }

    // 文件下载器
    static class FileDownloader implements Runnable {
        private final String fileURL;

        public FileDownloader(String fileURL) {
            this.fileURL = fileURL;
        }

        @Override
        public void run() {
            Debug.info("Downloading from " + fileURL);
            String fileName = fileURL.substring(fileURL.lastIndexOf('/') + 1);
            try {
                URL url = new URL(fileURL);
                String localFileName = System.getProperty("java.io.tmpdir")
                    + "/" + fileName;
                Debug.info("Saving to: " + localFileName);
                downloadFile(url, new FileOutputStream(localFileName), 1024);
            } catch (Exception e) {
```

```

        e.printStackTrace();
    }
    Debug.info("Done downloading from " + fileURL);
}

// 从指定的 URL 下载文件，并将其保存到指定的输出流中
private void downloadFile(URL url, OutputStream outputStream, int bufSize)
    throws MalformedURLException, IOException {
    // 完整代码参见本书配套下载资源
}
} // FileDownloader 结束
}

```

上述程序的 `main` 方法为其命令行参数中指定的 URL 列表中的每个 URL 创建一个线程。这些线程所执行的任务就是根据指定 URL 下载文件将其存储在本地磁盘中。而下载文件这个任务的处理逻辑我们是在 `Runnable` 接口的实现类 `FileDownloader` 的 `run` 方法中实现的。

从这个实例中看来，Java 多线程编程似乎很简单是吗？答案当然是“不是”。第 2 章我们会介绍 Java 多线程编程面临的一些挑战。

*1.9 多线程编程的优势和风险

多线程编程具有以下优势。

- 提高系统的吞吐率（Throughput）。多线程编程使得一个进程中可以有多个并发（Concurrent，即同时进行的）的操作。例如，当一个线程因为 I/O 操作而处于等待时，其他线程仍然可以执行其操作。
- 提高响应性（Responsiveness）。在使用多线程编程的情况下，对于 GUI 软件（如桌面应用程序）而言，一个慢的操作（比如从服务器上下载一个大的文件）并不会导致软件的界面出现被“冻住”的现象而无法响应用户的其他操作；对于 Web 应用程序而言，一个请求的处理慢了并不会影响其他请求的处理。
- 充分利用多核（Multicore）处理器资源。如今多核处理器的设备越来越普及，就算是手机这样的消费类设备也普遍使用多核处理器。实施恰当的多线程编程有助于我们充分利用设备的多核处理器资源，从而避免了资源浪费。
- 最小化对系统资源的使用。一个进程中的多个线程可以共享其所在进程所申请的资源（如内存空间），因此使用多个线程相比于使用多个进程进行编程来说，节约了对系统资源的使用。
- 简化程序的结构。线程可以简化复杂应用程序的结构。

多线程编程也有自身的问题与风险，包括以下几个方面。

- 线程安全（Thread Safe）问题。多个线程共享数据的时候，如果没有采取相应的并发访问控制措施¹⁴，那么就可能产生数据一致性问题，如读取脏数据（过期的数据）、丢失更新（某些线程所做的更新被其他线程所做的更新覆盖）等。
- 线程活性（Thread Liveness）问题。一个线程从其创建到运行结束的整个生命周期会经历若干状态。从单个线程的角度来看，RUNNABLE 状态是我们所期望的状态。但实际上，代码编写不当可能导致某些线程一直处于等待其他线程释放锁的状态（BLOCKED 状态）¹⁵，即产生了死锁（Deadlock）。例如，线程 T_1 拥有锁 L_1 ，并试图去获得锁 L_2 ，而此时线程 T_2 拥有锁 L_2 而试图去获得锁 L_1 ，这就导致线程 T_1 和 T_2 一直处于等待对方释放锁而一直又得不到锁的状态。当然，一直忙碌的线程也可能会出现问题，它可能面临活锁（Livelock）问题，即一个线程一直在尝试某个操作但就是无法进展，这就好比小猫一直追着自己的尾巴咬却一直也咬不到的情形。另外，线程是一种稀缺的计算资源，一个系统所拥有的处理器数量相比于该系统中存在的线程数量而言总是少之又少的。某些情况下可能出现线程饥饿（Starvation）的问题，即某些线程永远无法获取处理器执行的机会而永远处于 RUNNABLE 状态的 READY 子状态。
- 上下文切换（Context Switch）¹⁶。处理器从执行一个线程转向执行另外一个线程的时候操作系统所需要做的一个动作被称为上下文切换。由于处理器资源的稀缺性，因此上下文切换可以被看作多线程编程的必然副产物，它增加了系统的消耗，不利于系统的吞吐率。
- 可靠性。多线程编程一方面可以有利于可靠性，例如某个线程意外提前终止了，但这并不影响其他线程继续其处理。另一方面，线程是进程的一个组件，它总是存在于特定的进程中的，如果这个进程由于某种原因意外提前终止，比如某个 Java 进程由于内存泄漏导致 Java 虚拟机崩溃而意外终止，那么该进程中所有的线程也就随之而无法继续运行。因此，从提高软件可靠性的角度来看，某些情况下可能要考虑多进程多线程的编程方式¹⁷，而非简单的单进程多线程方式。

14 这个概念第 2 章会介绍。

15 锁的概念第 2 章会介绍，这里可暂且将其理解作为一种独占资源。

16 这个概念第 2 章会介绍。

17 例如，一个系统被分解为多个模块，每个模块是一个 Java 进程（程序）。各个模块间采用网络进行通信。

1.10 本章小结

本章介绍了线程、多线程编程这两个基本概念以及 Java 平台对线程的实现。本章知识结构如图 1-7 所示。

- 进程是程序的运行实例，一个进程可以包含多个线程，这些线程共享其所在进程的资源。
- 线程是进程中可独立执行的最小单位。Java 标准库类 `java.lang.Thread` 就是 Java 平台对线程的实现。特定线程总是在执行特定的任务，线程的 `run` 方法就是线程所要执行任务的处理逻辑的入口方法，该方法由 Java 虚拟机直接调用执行。Java 标准库接口 `java.lang.Runnable` 就是对任务的抽象，`Thread` 类就是 `Runnable` 接口的一个实现类。
- 应用程序负责线程的创建与启动，而线程调度器负责线程的调度和执行。Java 平台中有两种方式创建线程：创建 `Thread` 的子类和以 `Runnable` 接口实例为构造器参数直接通过 `new` 创建 `Thread` 实例。
- 在 Java 平台中，任何一段代码总是执行在确定的代码中的。同一段代码可以被不同的线程执行。代码可以通过 `Thread.currentThread()` 调用来获取其当前执行线程。
- 为每个线程设置一个简短而含义明确的名称属性有助于多线程程序的调试和问题定位。
- 一个线程从其创建到运行结束的整个生命周期会经历若干状态。线程执行过程中调用一些对象的方法（如 `Thread.sleep(long millis)`）或者执行特定的操作（如 I/O 操作）往往导致其状态的变更。线程转储是对线程进行监视的重要媒介。操作系统以及 JDK 都提供了一些工具（`jvisualvm`、`jstack` 和 `Java Mission Control`），可以用来获取线程转储。
- Java 平台是一个多线程的平台，线程的身影在 Java 平台中无处不在。按照线程间的创建关系，我们可以将多个线程间的关系理解为一个层次关系。Java 并无相关 API 用于获取一个线程的父线程或者子线程，父线程和子线程之间的生命周期并无必然联系。
- 线程是多线程编程的基本单位。多线程编程一方面有助于提高系统的吞吐率、提高软件的响应性、充分利用多核处理器资源、最小化对系统资源的使用和简化程序的结构，另一方面面临线程安全问题、线程活性问题、上下文切换和可靠性等问题。因此，多线程编程绝不仅仅是使用多个线程进行编程那么简单，多线程编程有其自身需要解决的问题，而这正是后续章节的主要内容。

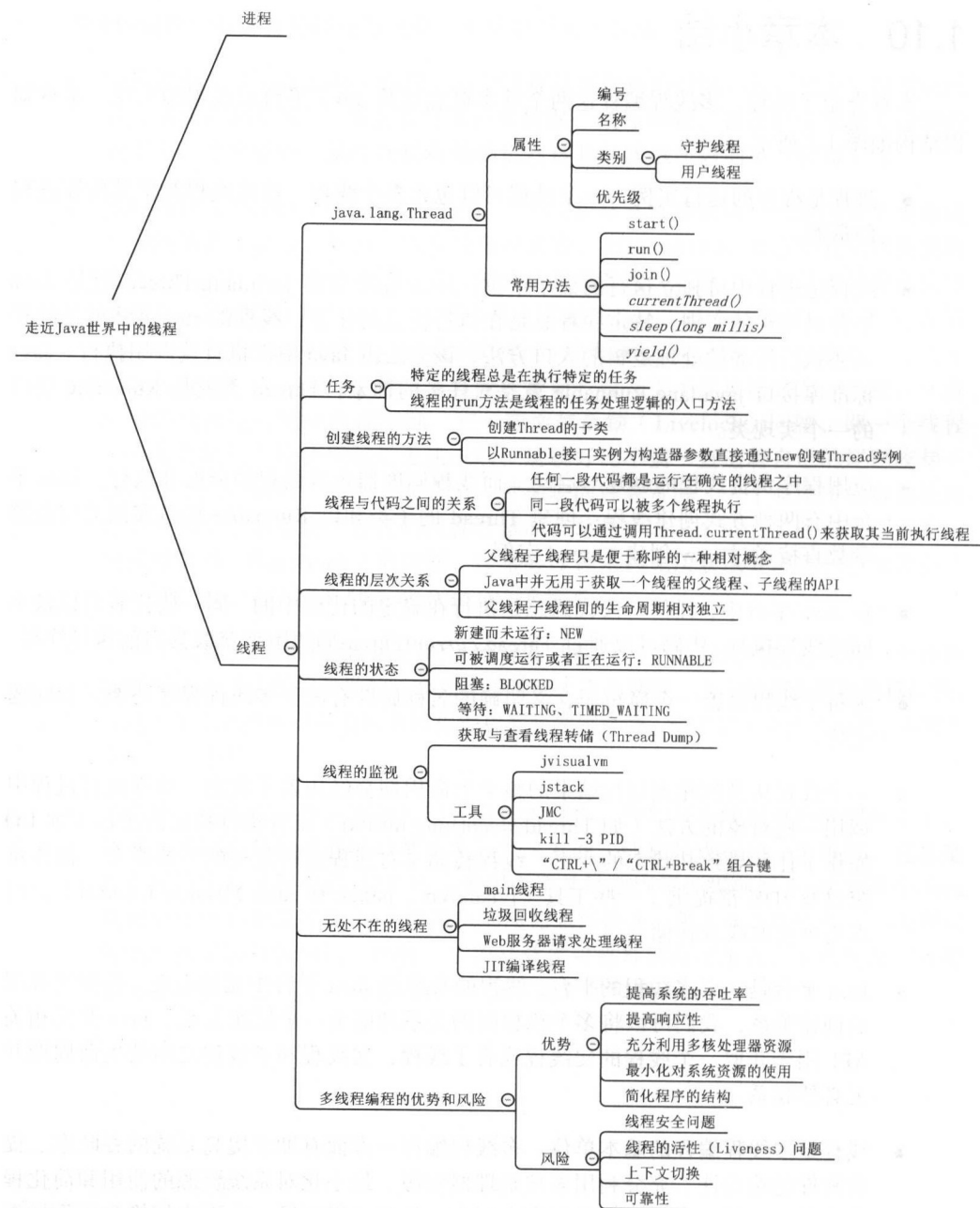


图 1-7 本章知识结构图

多线程编程的目标与挑战

If I had only one hour to save the world, I would spend fifty-five minutes defining the problem, and only five minutes finding the solution.

如果我只有 1 小时来拯救世界，我将花 55 分钟去定义这个问题而只花 5 分钟去寻找解决方案。

——Albert Einstein

本章通过一些基本概念讲解多线程编程的目标及其面临的挑战。这些概念是学习本书后续章节的基础，也是设计多线程程序和在实际工作中分析和定位多线程问题的基础。这些概念相对来说与具体的语言无关，即使用其他语言（如 C++）实现多线程编程也会涉及这些概念。另外，读者需要注意这些概念之间的联系。本章的扩展阅读内容会尽可能地列出一些读者深入理解相关概念所需要思考的问题，但是它们并不能代替读者自己去思考相关问题！

2.1 串行、并发与并行

假设我们有 3 件事情（事情 A、事情 B 和事情 C）要完成，完成每件事情所需的时间包括实际投入时间（如做些准备活动所需的时间）和等待的时间，完成这些事情所需的时间为：事情 A 耗时 15 分钟（实际投入 5 分钟，等待 10 分钟）、事情 B 耗时 10 分钟（实际投入 2 分钟，等待 8 分钟）、事情 C 耗时 10 分钟（实际投入 10 分钟，无等待耗时）。那么，我们有 3 种方式来完成这几件事情，如图 2-1 所示。

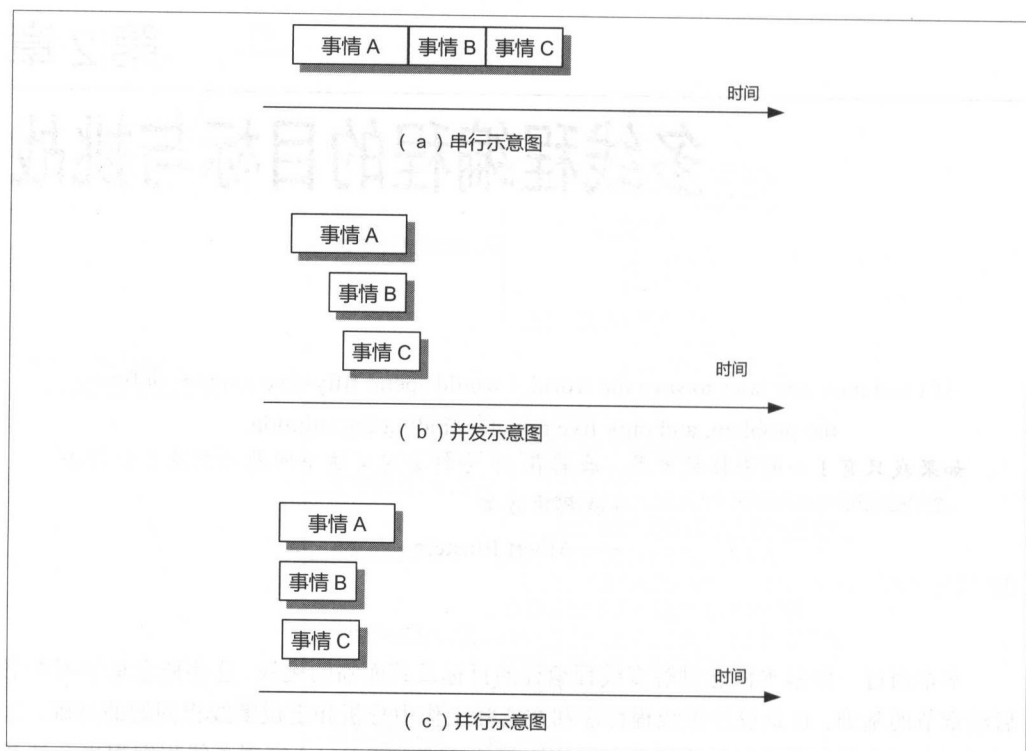


图 2-1 串行、并发与并行示意图

串行 (Sequential)，如图 2-1 (a) 所示。先开始做事情 A，待其完成之后再开始做事情 B，依次类推，直到事情 C 完成。这实际上顺序逐一完成几件事情，只需要投入一个人。在这种方式下 3 件事情总共耗时 35 (15+10+10) 分钟。

并发 (Concurrent)，如图 2-1 (b) 所示。这种方式也可以只投入一个人。这个人先开始做事情 A，事情 A 的准备活动做好后 (此时消耗了 5 分钟)，在等待事情 A 完成的这段时间内他开始做事情 B。为事情 B 的准备活动花了 2 分钟之后，在等待事情 B 完成的这段时间内他开始做事情 C，直到 10 分钟之后事情 C 完成。这整个过程实际上是以交替的方式利用等待某件事情完成的时间来做其他事情。在这种方式下 3 件事情总共耗时 17 (5+2+10) 分钟，这比第 1 种方式节约了一半多的时间。

并行 (Parallel)，如图 2-1 (c) 所示。这种方式需要投入 3 个人，每个人负责完成其中一件事情。这 3 个人在同一时刻开始齐头并进地完成这些事情。在这种方式下 3 件事情总共耗时 15 分钟 (取决于耗时最长的那件事情所需的时间)，比并发的方式节约了 2 分钟的时间。

可见，并发是串行的反面，并发往往可以提高我们对事情的处理效率，即一段时间内可以处理或者完成更多的事情。而并行是一种更为严格、理想的并发，即并行可以被看作并发的一个特例。串行好比多个车辆行驶在一股车道上，它们只能“鱼贯而行”。而并发好比多个车辆行驶在多股车道上，它们可以“并驾齐驱”。但是，行驶在多股车道上的车辆有时候也不得不“鱼贯而行”，比如进行施工的路段多股车道会被合并为一股小车道，从而使车辆只能“鱼贯而行”。因此，并发往往是带有部分串行的并发，而并发的极致就是并行（Parallel）。

从软件的角度来说，并发就是在一段时间内以交替的方式去完成多个任务，而并行就是以齐头并进的方式去完成多个任务。并发与上述生活中的并发并无实质的区别，不过二者还存在一些差异。首先，现实世界中的一个人可以以并发的方式去完成几件事情，而软件要以并发的方式去完成几个任务往往需要借助多个线程（而不是一个线程）。其次，软件世界中的并发也未必就比串行的处理效率更高或者效率提高得那么明显，这点在本书的后面内容中会体现出来。

从硬件的角度来说，在一个处理器一次只能够运行一个线程的情况下，由于处理器可以使用时间片（Time-slice）分配的技术来实现在同一段时间内运行多个线程，因此一个处理器就可以实现并发。而并行则需要靠多个处理器在同一时刻各自运行一个线程来呈现。

多线程编程的实质就是将任务的处理方式由串行改为并发，即实现并发化，以发挥并发的优势。而现实是以并发的方式对任务进行处理的过程也存在一些挑战，这点正是本章后续几节的主题。

如果一个任务的处理方式可以由串行改为并发（或者并行），那么我们就称这个任务是可并发化（或者可并行化）的。但是，有的任务的处理方式则可能必须是串行的。例如，在 Java 平台中读取一个文件就是串行的。

2.2 竞态

多线程编程中经常遇到的一个问题就是对于同样的输入，程序的输出有时候是正确的而有时候却是错误的。这种一个计算结果的正确性与时间有关的现象就被称为竞态（Race Condition）。

下面我们看一个竞态实例。某系统为了便于跟踪对其接收到的 HTTP 请求的处理，会为其收到的每个 HTTP 请求分配一个唯一编号（Request ID）。Request ID 是一个固定长度

的编码字符串，其中最后 3 位是一个在 0~999 循环递增（即从“000”递增到“999”，接着又从“000”开始递增）的序列号。我们很容易就可以写一个这样的 Request ID 生成器 RequestIDGenerator，如清单 2-1 所示。

清单 2-1 Request ID 生成器源码

```
public final class RequestIDGenerator implements CircularSeqGenerator {
    /**
     * 保存该类的唯一实例
     */
    private final static RequestIDGenerator INSTANCE = new RequestIDGenerator();
    private final static short SEQ_UPPER_LIMIT = 999;
    private short sequence = -1;

    // 私有构造器
    private RequestIDGenerator() {
        // 什么也不做
    }

    /**
     * 生成循环递增序列号
     *
     * @return
     */
    @Override
    public short nextSequence() {
        if (sequence >= SEQ_UPPER_LIMIT) {
            sequence = 0;
        } else {
            sequence++;
        }
        return sequence;
    }

    /**
     * 生成一个新的 Request ID
     *
     * @return
     */
    public String nextID() {
        SimpleDateFormat sdf = new SimpleDateFormat("yyMMddHHmmss");
        String timestamp = sdf.format(new Date());
        DecimalFormat df = new DecimalFormat("000");

        // 生成请求序列号
        short sequenceNo = nextSequence();
```

```

    return "0049" + timestamp + df.format(sequenceNo);
}

/**
 * 返回该类的唯一实例
 *
 * @return
 */
public static RequestIDGenerator getInstance() {
    return INSTANCE;
}
}

```

清单 2-2 模拟了 RequestIDGenerator 在实际环境（多线程环境）中的使用情况：每个业务线程（请求处理线程）在处理其接收到的请求前都要先为该请求申请一个 Request ID。

清单 2-2 竞态 Demo

```

public class RaceConditionDemo {

    public static void main(String[] args) throws Exception {
        // 客户端线程数
        int numberOfThreads = args.length > 0 ? Short.valueOf(args[0]) : Runtime
            .getRuntime().availableProcessors();
        Thread[] workerThreads = new Thread[numberOfThreads];
        for (int i = 0; i < numberOfThreads; i++) {
            workerThreads[i] = new WorkerThread(i, 10);
        }

        // 待所有线程创建完毕后，再一次性将其启动，以便这些线程能够尽可能地在同一时间内运行
        for (Thread ct : workerThreads) {
            ct.start();
        }
    }

    // 模拟业务线程
    static class WorkerThread extends Thread {
        private final int requestCount;

        public WorkerThread(int id, int requestCount) {
            super("worker-" + id);
            this.requestCount = requestCount;
        }

        @Override
        public void run() {
            int i = requestCount;
            String requestID;

```

```
RequestIDGenerator requestIDGen = RequestIDGenerator.getInstance();
while (i-- > 0) {
    // 生成 Request ID
    requestID = requestIDGen.nextID();
    processRequest(requestID);
}

// 模拟请求处理
private void processRequest(String requestID) {
    // 模拟请求处理耗时
    Tools.randomPause(50);
    System.out.printf("%s got requestID: %s %n",
        Thread.currentThread().getName(), requestID);
}
}
```

使用如下命令来模拟 4 个业务线程（其中每个线程处理 10 个请求）运行上述程序：

```
java io.github.viscent.mtia.ch2.RaceConditionDemo 4
```

从程序的输出上可以看出，有时候这个程序的输出正如我们期望的那样——每个请求对应的 Request ID 都不一样。而有的时候我们可以看到个别线程所获得的 Request ID 是重复的（尽管 RequestIDGenerator 的代码非常简单），如下面的输出：

```
worker-3 got requestID: 0049161101130503001
worker-2 got requestID: 0049161101130503001
worker-1 got requestID: 0049161101130503000
worker-1 got requestID: 0049161101130503003
worker-1 got requestID: 0049161101130503004
worker-0 got requestID: 0049161101130503001
// 此处省略部分输出
worker-3 got requestID: 0049161101130503037
```

可见，个别线程（worker-3、worker-2 和 worker-0）所获得的 Request ID 是相同的，并且这次模拟运行一共处理 40 个请求，因此 Request ID 中序列号部分最大的值应该是“039”，而实际只到“037”。

上述程序的输出有时候是正确的而有时候是错误的，可见该程序在多线程环境下运行出现了竞态。

2.2.1 二维表分析法：解释竞态的结果

术语
定义

状态变量 (State Variable)：即类的实例变量、静态变量。

共享变量 (Shared Variable)：即可以被多个线程共同访问的变量。共享变量中的“共享”强调的是“可以被共享”的可能性，因此称呼一个变量为共享变量并不表示该变量一定会被多个线程访问。状态变量由于可以被多个线程共享，因此也被称为共享变量。

上述例子中的竞态导致的结果是不同业务线程“拿到”了重复的 Request ID。这个结果说明 RequestIDGenerator (见清单 2-1) 中的 nextSequence()所返回的序列号重复了：Request ID 的前 4 位是一个固定值 (“0049”)，当中 12 位是系统的当前时间 (精确到秒)，最后 3 位是序列号。依照 nextSequence()方法实现的逻辑，序列号总是递增 (循环递增) 的，因此不同的业务线程即使是同一秒内申请 Request ID，它们所“拿到”的 Request ID 也会由于其中的序列号部分的不同而不同。由此可见，不同业务线程“拿到”重复的 Request ID 说明 nextSequence()方法所返回的序列号重复了。

可见，nextSequence()是导致上述竞态的直接因素。进一步来说，导致竞态的常见因素是多个线程在没有采取任何控制措施的情况下并发地更新、读取同一个共享变量。nextSequence()所访问的实例变量 sequence 就是这样一个例子：多个线程 (业务线程) 通过调用 nextSequence()并发地访问 sequence，显然这些线程没有采取任何控制措施。

nextSequence()中的语句“sequence++”看起来像是一个操作，它实际上相当于如下伪代码所表示的 3 个指令：

```
load(sequence,r1);// 指令①：将变量 sequence 的值从内存读到寄存器 r1
increment(r1);// 指令②：将寄存器 r1 的值增加 1
store(sequence,r1);// 指令③：将寄存器 r1 的内容写入变量 sequence 所对应的内存空间
```

因此，上述例子中的 4 个线程有可能以如表 2-1 所示的顺序交错运行。

表 2-1 竞态的二维表分析法示例

线程 时刻	worker-0	worker-2	worker-3	worker-1
t ₁	<未运行>	执行其他操作	执行其他操作	执行指令③
t ₂	执行其他操作	[sequence==0]执行指令①	[sequence==0] 执行指令①	[sequence==0] 执行其他操作
t ₃	执行其他操作	[r1==0]执行指令②	[r1==0]执行指令②	执行其他操作

续表

线程 时刻	worker-0	worker-2	worker-3	worker-1
t_4	[sequence==0] 执行指令①	[r1==1][sequence==0] 执行指令③	[r1==1][sequence==0] 执行指令③	<运行结束>
t_5	[r1==0] 执行指令②	[sequence==1] 执行其他操作	[sequence==1] 执行其他操作	<运行结束>
t_6	[r1==1] 执行指令③	<运行结束>	<运行结束>	<运行结束>
t_7	[sequence==1] 执行其他操作	<运行结束>	<运行结束>	<运行结束>

在这个交错运行顺序下，两个业务线程可能在同一时间读取到 `sequence` 的同一个值（见 t_3 时刻），一个业务线程对 `sequence` 所做的更新也可能“覆盖”其他线程对该变量所做的更新（见 t_6 时刻），这最终导致了各个业务线程“拿到”了重复的序列号，从而导致 Request ID 重复。具体分析如下。

在 t_1 时刻，worker-1 正在执行指令③，worker-0 未开始运行，worker-2 和 worker-3 都在执行其他操作。

在 t_2 时刻，worker-2 和 worker-3 同时各自执行指令①，此刻这两个线程都读取到上一时刻 worker-1 执行指令③的结果（即 `sequence` 被更新为 0）；worker-0 和 worker-1 在执行其他操作。

在 t_3 时刻，worker-2 和 worker-3 所在的处理器上的寄存器 `r1` 的值均为 0（上一时刻这两个线程各自执行指令①产生的结果），worker-2 和 worker-3 同时各自执行指令②；worker-0 和 worker-1 在执行其他操作。

在 t_4 时刻，worker-2、worker-3 所在的处理器上的寄存器 `r1` 的值均为 1（上一时刻这两个线程各自执行指令②产生的结果），worker-2、worker-3 同时各自执行指令③；worker-0 正在执行指令①，由于 worker-2、worker-3 对 `sequence` 的更新（通过执行指令③）正在此刻进行中（尚未完成），因此此刻 worker-0 能够读取到的 `sequence` 值仍然为 0；worker-1 已运行结束。

在 t_5 时刻，此刻 `sequence` 的值已经被 worker-2 和 worker-3 同时（在 t_4 时刻）更新为 1；由于 worker-0 在上一时刻执行指令①（读取 `sequence` 变量值）的时候 `sequence` 的“当

前值”依然是0，因此此刻 worker-0 所在的处理器上的寄存器 r1（worker-0 读取 sequence 的结果）的值为0（上一时刻该线程执行指令①产生的结果），可见，worker-0 读取到的是一个旧的 sequence 值，即这里产生了读取脏数据的问题，于是 worker-0 在其读到的旧数据的基础上执行指令②；worker-2、worker-3 在执行其他操作。

在 t_6 时刻，worker-0 所在的处理器上的寄存器 r1 的值为1（上一时刻该线程执行指令②产生的结果），在这个基础上 worker-0 执行指令③，即把 sequence 值更新为1。而这个更新会导致 worker-2 和 worker-3 各自在 t_4 时刻对 sequence 的更新（都将 sequence 更新为1）被“覆盖”。可见，这一刻 worker-0 执行的指令③会导致丢失更新（Update Lost），即其他线程对 sequence 所做的更新在后续时间里没有体现出来，这点从 t_7 时刻 sequence 仍然为1可以看出来——4个线程各自都执行将 sequence 值增加1的逻辑（sequence++），最终体现到结果上仅仅是将 sequence 增加2（从-1变为1），而不是我们期望的4。

可见，worker-0、worker-2 和 worker-3 最终“拿到”的序列号都是“001”，从而导致这些线程最终获得的 Request ID 也是重复的。这个重复是读取脏数据（发生在 t_5 时刻）、丢失更新（发生在 t_6 时刻和 t_5 时刻）导致的。而上述输出的 Request ID 中序列号部分最大的值应该是“037”（而不是期望的“039”）正是丢失更新的结果。另外，如果线程执行的交错顺序不是表 2-1 那样的，那么清单 2-2 的 Demo 的运行结果可能又是另外一种。

根据上述分析我们可以更进一步来定义竞态：竞态（Race Condition）是指计算的正确性依赖于相对时间顺序（Relative Timing）或者线程的交错（Interleaving）。根据这个定义可知，竞态不一定就导致计算结果的不正确，它只是不排除计算结果时而正确时而错误的可能。例如，上述例子中各个线程如果不像表 2-1 那样的交错顺序执行“sequence++”这个操作，而恰好是各个线程以先后顺序执行“sequence++”，那么这些线程所获得的 Request ID 就可能是正确的。

竞态往往伴随着读取脏数据（Dirty Read）问题，即线程读取到一个过时的数据、丢失更新（Lost Update）问题，即一个线程对数据所做的更新没有体现在后续其他线程对该数据的读取上。而上述的二维表分析法是分析竞态问题的一种简单而有效的方法。

注意

竞态不一定就导致计算结果的不正确，它只是不排除计算结果时而正确时而错误的可能。

2.2.2 竞态的模式与竞态产生的条件

从上述竞态典型实例（见清单 2-2）中我们可以提炼出竞态的两种模式：read-modify-

write（读一改一写）和 check-then-act（检测而后行动）。

read-modify-write（读一改一写）操作，该操作可以被细分为这样几个步骤：读取一个共享变量的值（read），然后根据该值做一些计算（modify），接着更新该共享变量的值（write）。例如，在清单 2-1 中，nextSequence()中的“sequence++”就是 read-modify-write 模式的一个实例。“sequence++”实际上相当于如下伪代码表示的几个指令的组合。

```
load(sequence, r1); // 指令①read: 从内存将 sequence 的值读到寄存器 r1（读取共享变量值）
increment(r1); // 指令② modify: 将寄存器 r1 的值增加 1（根据共享变量值做一些计算）
store(sequence, r1); // 指令③ write: 将寄存器 r1 的内容写入 sequence 对应的内存空间（更新共享变量）
```

一个线程在执行完指令①之后到开始（或者正在）执行指令②的这段时间内其他线程可能已经更新了共享变量（sequence）的值，这就使得该线程在执行指令②时使用的是共享变量的旧值（读脏数据）。接着，该线程把根据这个旧值计算出来的结果更新到共享变量，而这又使得其他线程对该共享变量所做的更新被“覆盖”，即造成了更新丢失。读者也可以根据二维表分析法自行分析多个线程并发执行上述代码的时候可能导致丢失更新和读脏数据的问题。

check-then-act（检测而后行动）操作，该操作可以被细分为这样几个步骤：读取某个共享变量的值，根据该变量的值决定下一步的动作是什么。例如，在清单 2-1 中，nextSequence()中的 if-else 语句就是该模式的一个实例。

```
if (sequence >= 999) { // 子操作①check: 检测共享变量的值
    sequence = 0; // 子操作②act: 下一步的操作
} else {
    sequence++;
}
```

一个线程在执行完子操作①到开始（或者正在）执行子操作②的这段时间内，其他线程可能已经更新了共享变量的值而使得 if 语句中的条件变为不成立，那么此时该线程仍然会执行子操作②，尽管这个子操作所需的前提（if 语句中的条件）实际上并未成立！读者也可以根据二维表分析法自行分析多个线程并发执行上述代码的时候可能导致丢失更新和读脏数据的问题。

从上述分析中我们可以总结出竞态产生的一般条件。设 O_1 和 O_2 是并发访问共享变量 V 的两个操作，这两个操作并非都是读操作。如果一个线程在执行 O_1 期间（开始执行而未执行结束）另外一个线程正在执行 O_2 ，那么无论 O_2 是在读取还是更新 V 都会导致竞态。从这个角度来看，竞态可以被看作访问（读取、更新）同一组共享变量的多个线程所执行的操作相互交错（Interleave），比如一个线程读取共享变量并以该共享变量为基础进行计

算的期间另外一个线程更新了该共享变量的值而导致的干扰（读取脏数据）或者冲突（丢失更新）的结果。

对于局部变量（包括形式参数和方法体内定义的变量），由于不同的线程各自访问的是各自的那一份局部变量，因此局部变量的使用不会导致竞态！例如，如清单 2-3 所示的 `nextSequence()` 的方法体同时具备了 `check-then-act` 模式与 `read-modify-write` 模式的代码结构，但是由于其使用的变量 `sequence` 是一个局部变量（形式参数），因此它不会导致竞态。

清单 2-3 不会出现竞态的一个例子

```
public class NoRaceCondition {

    public int nextSequence(int sequence) {

        // 以下语句使用的是局部变量而非状态变量，并不会产生竞态
        if (sequence >= 999) {
            sequence = 0;
        } else {
            sequence++;
        }
        return sequence;
    }

}
```

如清单 2-2 所示 Demo 中的竞态问题，其中一个解决方法就是在 `RequestIDGenerator.nextSequence()` 的声明中添加一个 `synchronized` 关键字，如清单 2-4 所示。

清单 2-4 不会导致竞态的序列号生成器源码

```
public class SafeCircularSeqGenerator implements CircularSeqGenerator {
    private short sequence = -1;

    public synchronized short nextSequence() {
        if (sequence >= 999) {
            sequence = 0;
        } else {
            sequence++;
        }
        return sequence;
    }
}
```

`synchronized` 关键字会使其修饰的方法在任一时刻只能够被一个线程执行，这使得该方法涉及的共享变量在任一时刻只能够有一个线程访问（读、写），从而避免了这个方法的交错执行而导致的干扰，这样就消除了竞态。第 3 章会详细解释 `synchronized`。

2.3 线程安全性

如果我们使用如下命令以单线程环境（业务线程只有一个）来运行如清单 2-2 所示的 Demo：

```
java io.github.viscent.mtia.ch2.RaceConditionDemo 1
```

那么，我们可以发现该程序的输出总是正确（符合我们的期望）的：

```
worker-0 got requestID: 0049161103165501000
worker-0 got requestID: 0049161103165501001
worker-0 got requestID: 0049161103165501002
worker-0 got requestID: 0049161103165501003
worker-0 got requestID: 0049161103165501004
worker-0 got requestID: 0049161103165501005
worker-0 got requestID: 0049161103165501006
worker-0 got requestID: 0049161103165501007
worker-0 got requestID: 0049161103165501008
worker-0 got requestID: 0049161103165501009
```

正如我们在 2.2 节中看到的那样，在多线程环境下上述程序会出现竞态——程序输出的结果时而正确时而错误；而且输出是错误的话，具体的错误还不完全一样。

一般而言，如果一个类在单线程环境下能够运作正常，并且在多线程环境下，在其使用方不必为其做任何改变的情况下也能运作正常，那么我们就称其是线程安全（Thread-safe）的，相应地我们称这个类具有线程安全性（Thread Safety）。反之，如果一个类在单线程环境下运作正常而在多线程环境下则无法正常运作，那么这个类就是非线程安全的。清单 2-1 中的序号生成器就是非线程安全的。因此，一个类如果能够导致竞态，那么它就是非线程安全的；而一个类如果是线程安全的，那么它就不会导致竞态。

使用一个类的时候我们必须先弄清楚这个类是否是线程安全的。因为这关系到我们如何正确使用这些类。这好比微波炉加热食物前我们必须先弄清楚所用的容器是否适宜进行微波加热（金属容器不宜微波加热）。Java 标准库中的一些类如 ArrayList、HashMap 和 SimpleDateFormat，都是非线程安全的，在多线程环境下直接使用它们可能导致一些非预期的结果，甚至是一些灾难性的结果。比如，多线程环境下多个线程共享同一个 HashMap 实例（而不采取任何控制措施）可能导致死循环（表现为主机上的某个处理器使用率一直为 100%）和内存泄漏（最后可能导致 Java 虚拟机崩溃）。一般来说，Java 标准库中的类在其 API 文档（JavaDoc）中会说明其是否是线程安全的（没有说明其是否是线程安全的，则可能是也可能不是线程安全的）。

从线程安全的定义上我们不难看出,如果一个线程安全的类在多线程环境下能够正常运作,那么它在单线程环境下也能正常运作。既然如此,那为什么不干脆把所有的类都做成线程安全的呢?是否将一个类做成线程安全的,从某种程度上来说是一个设计上的权衡的结果或决定:一方面,一个类是否需要是线程安全的与这个类预期被使用的方式有关,比如,我们希望一个类总是只能被一个线程独自使用,那么就没有必要将这个类做成线程安全的¹。其次,把一个类做成线程安全的往往是有额外代价的。

一个类如果不是线程安全的,我们就说它在多线程环境下直接使用存在线程安全问题。线程安全问题概括来说表现为3个方面:原子性、可见性和有序性。

2.4 原子性

原子 (Atomic) 的字面意思是不可分割的 (Indivisible)。对于涉及共享变量访问的操作,若该操作从其执行线程以外的任意线程来看是不可分割的,那么该操作就是原子操作,相应地我们称该操作具有原子性 (Atomicity)。

许多资料都会提及原子操作的定义中的“不可分割”,但是很少有资料会对其含义做进一步的解释。而弄清楚“不可分割”的具体含义是理解原子性的关键所在。所谓“不可分割”,其中一个含义是指访问(读、写)某个共享变量的操作从其执行线程以外的任何线程来看,该操作要么已经执行结束要么尚未发生,即其他线程不会“看到”该操作执行了部分的中间效果。

在生活中我们可以找到的一个原子操作的例子就是人们从 ATM 机提取现金:尽管从 ATM 软件的角度来说,一笔取款交易涉及扣减户主账户余额、吐钞器吐出钞票、新增交易记录等一系列操作,但是从用户(我们)的角度来看 ATM 取款就是一个操作。该操作要么成功了,即我们拿到现金(户主账户的余额会被扣减),这个操作发生过了;要么失败了,即我们没有拿到现金,这个操作就像从来没有发生过一样(当然,户主账户的余额也不会被扣减)。除非 ATM 软件有缺陷,否则我们不会遇到吐钞口吐出部分现金而我们的账户余额却被扣除这样的部分结果。在这个例子中,户主账户余额就相当于我们所说的共享变量,而 ATM 机及其用户(人)就分别相当于上述定义中原子操作的执行线程和其他线程。

下面通过一个例子来体会一下“不可分割”的含义。如清单 2-5 所示,假设线程 T_1 通过执行 `updateHostInfo` 方法来更新主机信息 (HostInfo), 线程 T_2 则通过执行

¹ 这里假设这个类引用的其他对象同样也不会被其他线程访问。

`connectToHost` 方法来读取主机信息，并据此与相应的主机建立网络连接。那么，`updateHostInfo` 方法中的操作（更新主机 IP 地址和端口号）必须是一个原子操作，即这个操作必须是“不可分割”的。否则，可能出现这样的情形：假设 `hostInfo` 的初始值表示的是 IP 地址为“192.168.1.101”、端口号为 8081 的主机， T_1 执行 `updateHostInfo` 方法试图将 `hostInfo` 更新为 IP 地址为“192.168.1.100”、端口号为 8080 的主机的时候， T_2 可能刚好执行 `connectToHost` 方法，那么此时由于 T_1 可能刚刚执行完语句①而未开始语句②（即只更新完 IP 地址而尚未更新端口号），因此 T_2 可能读取到 IP 地址为“192.168.1.100”而端口号却仍然为 8081 的主机信息，即 T_2 读取到了一个错误的主机信息（IP 地址为“192.168.1.100”的主机上面并没有开启侦听端口 8081，它开启的是 8080），从而无法建立网络连接！这里的错误是由于 `updateHostInfo` 方法中的操作不是原子操作（不具备“不可分割”的特性）而使其他线程读取了脏数据（错误的主机信息）导致的。

清单 2-5 原子操作问题示例

```
public class AtomicExample {
    private HostInfo hostInfo;

    public void updateHostInfo(String ip, int port) {
        // 以下操作不是原子操作
        hostInfo.setIp(ip); // 语句①
        hostInfo.setPort(port); // 语句②
    }

    public void connectToHost() {
        String ip = hostInfo.getIp();
        int port = hostInfo.getPort();
        connectToHost(ip, port);
    }

    private void connectToHost(String ip, int port) {
        // ...
    }

    public static class HostInfo {
        private String ip;
        private int port;
        //...
    }
}
```

设 O_1 和 O_2 是访问共享变量 V 的两个原子操作，这两个操作并非都是读操作。那么一个线程执行 O_1 期间（开始执行而未执行完毕），其他线程无法执行 O_2 。也就是说，访问同一组共享变量的原子操作是不能够被交错的，这就排除了一个线程执行一个操作期间另

外一个线程读取或者更新该操作所访问的共享变量而导致的干扰（读脏数据）和冲突（丢失更新）的可能。这就是“不可分割”的第二个含义。由此可见，使一个操作具备原子性也就消除了这个操作导致竞态的可能性。

理解原子操作这个概念还需要注意以下两点。

- 原子操作是针对访问共享变量的操作而言的。也就是说，仅涉及局部变量访问的操作无所谓是否是原子的，或者干脆把这一类操作都看成原子操作，例如，在清单 2-3 中，`nextSequence(int sequence)` 中的操作就是这样一类操作。
- 原子操作是从该操作的执行线程以外的线程来描述的，也就是说它只有在多线程环境下有意义。换言之，单线程环境下一个操作无所谓是否具有原子性，或者我们干脆把这一类操作都看成原子操作。

提示

原子操作多线程环境下的一个概念，它是针对访问共享变量的操作而言的。原子操作的“不可分割”包括以下两层含义。

- 访问（读、写）某个共享变量的操作从其执行线程以外的任何线程来看，该操作要么已经执行结束要么尚未发生，即其他线程不会“看到”该操作执行了部分的中间效果。
- 访问同一组共享变量的原子操作是不能够被交错的。

总的来说，Java 中有两种方式来实现原子性。一种是使用锁（Lock）。锁具有排他性，即它能够保障一个共享变量在任意一个时刻只能被一个线程访问。这就排除了多个线程在同一时刻访问同一个共享变量而导致干扰与冲突的可能，即消除了竞态。另一种是利用处理器提供的专门 CAS（Compare-and-Swap）指令，CAS 指令实现原子性的方式与锁实现原子性的方式实质上是相同的，差别在于锁通常是在软件这一层次实现的，而 CAS 是直接硬件（处理器和内存）这一层次实现的，它可以被看作“硬件锁”。

在 Java 语言中，`long` 型和 `double` 型以外的任何类型的变量的写操作都是原子操作，即对基础类型（`long/double` 除外，仅包括 `byte`、`boolean`、`short`、`char`、`float` 和 `int`）的变量和引用型变量的写操作都是原子的。这点由 Java 语言规范（JLS, Java Language Specification）规定，由 Java 虚拟机具体实现。

对 `long/double` 型变量的写操作由于 Java 语言规范并不保障其具有原子性²，因此在多

2 参见 Java 语言规范“17.7. Non-Atomic Treatment of double and long”：<http://docs.oracle.com/javase/specs/jls/se8/html/jls-17.html#jls-17.7>。

个线程并发访问同一 long/double 型变量的情况下，一个线程可能会读取到其他线程更新该变量的“中间结果”。例如，设一个 long 型共享变量 value 的初始值为 0，有两个线程（updateThread1、updateThread2）并发地分别将 value 更新为-1 和 0，另外一个线程（main）会读取 value 的值，如清单 2-6 所示。

清单 2-6 long/double 型变量写操作的非原子 Demo

```
/**
 * 本 Demo 必须使用 32 位 Java 虚拟机才能看到非原子操作的效果。 <br>
 * 运行本 Demo 时也可以指定虚拟机参数“-client”
 *
 * @author Viscent Huang
 */
public class NonAtomicAssignmentDemo implements Runnable {
    static long value = 0;
    private final long valueToSet;

    public NonAtomicAssignmentDemo(long valueToSet) {
        this.valueToSet = valueToSet;
    }

    public static void main(String[] args) {
        // 线程 updateThread1 将 data 更新为 0
        Thread updateThread1 = new Thread(new NonAtomicAssignmentDemo(0L));
        // 线程 updateThread2 将 data 更新为-1
        Thread updateThread2 = new Thread(new NonAtomicAssignmentDemo(-1L));

        PrintStream ps = new PrintStream(new OutputStream() {
            @Override
            public void write(int b) throws IOException {
                // 不实际进行输出
            }
        });

        updateThread1.start();
        updateThread2.start();

        // 共享变量 value 的快照（即瞬间值）
        long snapshot;
        while (0 == (snapshot = value) || -1 == snapshot) {
            /*
             * 这里，我们将“snapshot = value”放在循环内是为了不断地读取共享变量 value 的值。
             * 而 server 运行模式的 Java 虚拟机（的 JIT 编译器）在生成这段代码对应的机器码的时候，
             * 可能会进行循环优化（循环不变表达式外提）而将“snapshot = value”对应的机器码
             * 放在这个循环语句之外，这就使我们无法重复读取 value 的值。
             * 下面的输出语句并不是为了输出数据，而是为了防止 JIT 编译器做“循环不变表达式外提”这种优化。
             * 或者，将以下语句注释掉，以 client 模式运行本 Demo 也可以达到同样的效果——
            */
        }
    }
}
```

```

    * 防止“循环不变表达式外提”优化。
    *
    * 循环不变表达式外提 (Loop-invariant code motion):
    * http://www.compileroptimizations.com/category/hoisting.htm
    */
    ps.print(snapshot);
}

System.err.printf("Unexpected data: %d(0x%016x)", snapshot, snapshot);
ps.close();
System.exit(0);
}

@Override
public void run() {
    for (;;) {
        value = valueToSet;
    }
}
}

```

使用 32 位（而不是 64 位）Java 虚拟机运行如清单 2-6 所示的 Demo，我们可以看到该程序的输出是：

```
Unexpected data: 4294967295(0x00000000ffffffff)
```

或者

```
Unexpected data: -4294967296(0xffffffff00000000)
```

可见，main 线程读取到共享变量 value 的值可能既不是 0（对应无符号十六进制数 0x0000000000000000）也不是 -1（对应无符号十六进制数 0xffffffffffffff），而是其他两个线程更新 value 时的“中间结果”——4294967295（对应无符号十六进制数 0x00000000ffffffff）或者 -4294967296（对应无符号十六进制数 0xffffffff00000000），即一个线程对 value 变量的低 32 位（Lower 32-bits，4 字节）更新与另外一个线程对 value 变量的高 32 位（Higher 32-bits，4 字节）更新所“混合”出来的一个结果³！

尽管如此，Java 语言规范特别地规定对于 volatile 关键字修饰的 long/double 型变量的

3 这个结果的出现是因为 Java 中的 long/double 型变量会占用 64 位（8 字节）的存储空间，而 32 位的 Java 虚拟机对这种变量的写操作可能会被分解为两个步骤来实施，比如先写低 32 位，再写高 32 位。那么，在多个线程试图共享同一个这样的变量时就可能出现一个线程在写高 32 位的时候，另外一个线程正在写低 32 位的情形。

写操作具有原子性。因此，我们只需要用 `volatile` 关键字修饰清单 2-6 中的共享变量 `value`，就可以保障对该变量的写操作的原子性。

`volatile long value;` // 通过使用 `volatile` 关键字使对变量 `value` 的写操作具有原子性

`volatile` 关键字仅能够保障变量写操作的原子性，它并不能保障其他操作（比如 `read-modify-write` 操作和 `check-then-act` 操作）的原子性。第 3 章会进一步介绍该关键字。

利用 Java 语言对变量（`long/double` 型变量除外）写操作的原子性的保障，清单 2-5 中的原子操作问题只需要通过改写 `updateHostInfo` 方法就可以解决：

```
public void updateHostInfo2(String ip, int port) {  
    HostInfo newHostInfo = new HostInfo(ip, port);  
    hostInfo = newHostInfo; // 原子操作  
}
```

Java 语言中针对任何变量的读操作都是原子操作。

从原子操作的“不可分割”特性可知，使一个操作具有原子性就可以消除该操作导致竞态的可能性。因此，我们可以将 `read-modify-write` 操作和 `check-then-act` 操作转换为原子操作来消除竞态。

竞态模式中的 `read-modify-write` 操作本身不是原子操作，但是我们可以使用 Java 语言提供的机制使其具有原子性。例如，如清单 2-4 所示的代码就是通过使用 `synchronized` 关键字使得如清单 2-1 所示的代码中的 `read-modify-write` 操作转换为原子操作的。

竞态模式中的 `check-then-act` 操作本身不是原子操作。同样地，我们也可以使用与将 `read-modify-write` 操作转换为原子操作同样的方法将这种操作转换为原子操作，即使其具有原子性。

扩展阅读 原子操作+原子操作=原子操作？

这个问题我们通过一个实例就可以得到答案。例如，对于共享 `int` 型变量 `a` 和 `b`（初始值皆为 0），假设线程 A 执行如下操作：

```
a=1; // 语句①  
b=2; // 语句②
```

显然，语句①和语句②都是原子操作（这点由 Java 语言规范保证）。但是，在线程 A 执行完语句①之后和在执行语句②之前的这一刻，另外一个线程 B 可以读取变量 `a` 和变量 `b` 的值。那么，此刻线程 B 读取到变量 `a` 和变量 `b` 的值分别为 1 和 0，也就是说它读取到

了线程 A 所执行操作的中间结果，这有悖于原子操作不可分割的特性。因此，“原子操作+原子操作”所得到的复合操作并非原子操作。

2.5 可见性

在多线程环境下，一个线程对某个共享变量进行更新之后，后续访问该变量的线程可能无法立刻读取到这个更新的结果，甚至永远也无法读取到这个更新的结果。这就是线程安全问题的另外一个表现形式：可见性（Visibility）。

如果一个线程对某个共享变量进行更新之后，后续访问该变量的线程可以读取到该更新的结果，那么我们就称这个线程对该共享变量的更新对其他线程可见，否则我们就称这个线程对该共享变量的更新对其他线程不可见。可见性就是指一个线程对共享变量的更新的结果对于读取相应共享变量的线程而言是否可见的问题。多线程程序在可见性方面存在问题意味着某些线程读取到了旧数据（Stale Data），而这可能导致程序出现我们所不期望的结果。

下面我们看一个可见性问题的 Demo。假设我们要执行一个比较耗时的任务，如果这个任务在指定时间内仍然没有执行结束，那么我们就主动将其取消，如清单 2-7 所示。使用如下命令以 server 虚拟机模式运行该 Demo：

```
java -server io.github.viscent.mtia.ch2.VisibilityDemo
```

上述程序可能一直在运行⁴，而不是我们所期望的程序运行 10 秒（左右）之后就输出“Task was canceled.”而停止。这说明，main 线程执行 TimeConsumingTask 的 cancel 方法并没有达到其目标——使子线程 thread 停止。这种现象只有一种解释，那就是子线程 thread 所读取到的 toCancel 变量值始终是 false，尽管某个时刻 main 线程会将共享变量 toCancel 的值更新为 true。可见，这里产生了可见性问题，即 main 线程对共享变量 toCancel 的更新对子线程 thread 而言不可见。

清单 2-7 可见性问题 Demo

```
public class VisibilityDemo {

    public static void main(String[] args) throws InterruptedException {
        TimeConsumingTask timeConsumingTask = new TimeConsumingTask();
        Thread thread = new Thread(new TimeConsumingTask());
```

4 这里需要注意，该程序只有在 Java 虚拟机以 server 模式（而不是 client 模式）运行的情况下才会一直运行而不结束。这就是我们在相应的命令中添加“-server”参数的原因。


```

        thread.start();

        // 指定的时间内任务没有执行结束的话，就将其取消
        Thread.sleep(10000);
        timeConsumingTask.cancel();
    }
}

class TimeConsumingTask implements Runnable {
    private boolean toCancel = false;

    @Override
    public void run() {
        while (!toCancel) {
            if (doExecute()) {
                break;
            }
        }
        if (toCancel) {
            System.out.println("Task was canceled.");
        } else {
            System.out.println("Task done.");
        }
    }

    private boolean doExecute() {
        boolean isDone = false;
        System.out.println("executing...");

        // 模拟实际操作的时间消耗
        Tools.randomPause(50);
        // 省略其他代码

        return isDone;
    }

    public void cancel() {
        toCancel = true;
        System.out.println(this + " canceled.");
    }
}

```

上述例子中的可见性问题是因為代碼沒有給 JIT 編譯器足夠的提示而使得其認為狀態變量 `toCancel` 只有一個線程對其進行訪問，從而導致 JIT 編譯器為了避免重複讀取狀態變量 `toCancel` 以提高代碼的運行效率，而將 `TimeConsumingTask` 的 `run` 方法中的 `while` 循環

优化成与如下代码等效的本地代码（机器码）⁵：

```
if (!toCancel) {
    while (true) {
        if (doExecute()) {
            break;
        }
    }
}
```

不幸的是，此时这种优化导致了死循环，也就是我们所看到的程序一直运行而没有退出。

另一方面，可见性问题与计算机的存储系统有关。程序中的变量可能会被分配到寄存器（Register）而不是主内存中进行存储。每个处理器都有其寄存器，而一个处理器无法读取另外一个处理器上的寄存器中的内容。因此，如果两个线程分别运行在不同的处理器上，而这两个线程所共享的变量却被分配到寄存器上进行存储，那么可见性问题就会产生。另外，即便某个共享变量是被分配到主内存中进行存储的，也不能保证该变量的可见性。这是因为处理器对主内存的访问并不是直接访问，而是通过其高速缓存（Cache）子系统进行的。一个处理器上运行的线程对变量的更新可能只是更新到该处理器的写缓冲器（Store Buffer）中，还没有到达该处理器的高速缓存中，更不用说到达主内存中了。而一个处理器的写缓冲器中的内容无法被另外一个处理器读取，因此运行在另外一个处理器上的线程无法看到这个线程对某个共享变量的更新。即便一个处理器上运行的线程对共享变量的更新结果被写入该处理器的高速缓存，由于该处理器将这个变量更新的结果通知给其他处理器的时候，其他处理器可能仅仅将这个更新通知的内容存入无效化队列（Invalidate Queue）中，而没有直接根据更新通知的内容更新其高速缓存的相应内容，这就导致了其他处理器上运行的其他线程后续再读取相应共享变量时，从相应处理器的高速缓存中读取到的变量值是一个过时的值。

术语 定义

处理器并不是直接与主内存（RAM）打交道而执行内存的读、写操作，而是通过寄存器（Register）、高速缓存（Cache）、写缓冲器（Store Buffer，也称 Write Buffer）和无效化队列（Invalidate Queue）等部件执行内存的读、写操作的。从这个角度来看，这些部件相当于主内存的副本，因此本书为了叙述方便将这些部件统称为处理器对主内存的缓存，简称处理器缓存。

虽然一个处理器的高速缓存中的内容不能被另外一个处理器直接读取，但是一个处理

⁵ 这种优化被称为循环不变表达式外提（Loop-invariant Code Motion），也称循环提升（Loop Hoisting）。

器可以通过缓存一致性协议（Cache Coherence Protocol）来读取其他处理器的高速缓存中的数据，并将读到的数据更新到该处理器的高速缓存中。这种一个处理器从其自身处理器缓存以外的其他存储部件中读取数据并将其反映（更新）到该处理器的高速缓存的过程，我们称之为缓存同步。相应地，我们称这些存储部件的内容是可同步的，这些存储部件包括处理器的高速缓存、主内存。缓存同步使得一个处理器（上运行的线程）可以读取到另外一个处理器（上运行的线程）对共享变量所做的更新，即保障了可见性。因此，为了保障可见性，我们必须使一个处理器对共享变量所做的更新最终被写入该处理器的高速缓存或者主内存中（而不是始终停留在其写缓冲器中），这个过程被称为冲刷处理器缓存。并且，一个处理器在读取共享变量的时候，如果其他处理器在此之前已经更新了该变量，那么该处理器必须从其他处理器的高速缓存或者主内存中对相应的变量进行缓存同步。这个过程被称为刷新处理器缓存。因此，可见性的保障是通过使更新共享变量的处理器执行冲刷处理器缓存的动作，并使读取共享变量的处理器执行刷新处理器缓存的动作来实现的。

那么，在 Java 平台中我们如何保证可见性呢？这里我们先举一个例子，对于如清单 2-7 所示的代码，我们只需要在 `TimeConsumingTask` 的实例变量 `toCancel` 的声明中添加一个 `volatile` 关键字即可，即将其声明改为如下语句即可保证可见性：

```
private volatile boolean toCancel = false;
```

这里，`volatile` 关键字所起到的一个作用就是，提示 JIT 编译器被修饰的变量可能被多个线程共享，以阻止 JIT 编译器做出可能导致程序运行不正常的优化。另外一个作用就是读取一个 `volatile` 关键字修饰的变量会使相应的处理器执行刷新处理器缓存的动作，写一个 `volatile` 关键字修饰的变量会使相应的处理器执行冲刷处理器缓存的动作，从而保障了可见性。第 3 章会详细讲解 `volatile` 关键字。

可见性得以保障，并不意味着一个线程能够看到另外一个线程更新的所有变量的值。如果一个线程在某个时刻更新了多个共享变量的值，那么此后其他线程再来读取这些变量时，这些线程所读取到的变量值有些是其他线程更新过的值，而有些则可能仍然是其他线程更新之前的值（旧值）。

另外，由于某些处理器（如常见的 x86 处理器）在可见性方面足够“强大”，加上实际工作中我们能够接触到的处理器类型非常有限，因此可见性问题往往不是必然出现的。尽管可见性问题不是必然出现的，但是这绝不意味着我们可以抱着侥幸心理而无视它！这就好比飞机是一种比较安全的交通工具，空难出现的概率比起其他交通事故出现的概率要低得多，但是空难一旦发生，则往往意味着没有生还者！

约定 对于同一个共享变量而言，一个线程更新了该变量的值之后，其他线程能够读取到这个更新后的值，那么这个值就被称为该变量的相对新值。如果读取这个共享变量的线程在读取并使用该变量的时候其他线程无法更新该变量的值，那么该线程读取到的相对新值就被称为该变量的最新值。

可见性的保障仅仅意味着一个线程能够读取到共享变量的相对新值，而不能保障该线程能够读取到相应变量的最新值。

扩展阅读 单处理器系统是否存在可见性问题？

可见性问题是多线程衍生出来的问题，它与程序的目标运行环境是单处理器（Uni-processor）还是多核处理器无关。也就是说，单处理器系统中实现的多线程编程也可能出现可见性问题。

在目标运行环境是单处理器的情况下，多线程的并发执行实际上是通过时间片（Time Slice）分配实现的。此时，虽然多个线程是运行在同一个处理器上的，但是由于在发生上下文切换（Context Switch，参见 2.7 节）的时候，一个线程对寄存器（Register）变量的修改会被作为该线程的线程上下文保存起来，这导致另外一个线程无法“看到”该线程对这个变量的修改，因此，单处理器系统中实现的多线程编程也可能出现可见性问题。

扩展阅读 可见性与原子性的联系与区别

原子性描述的是一个线程对共享变量的更新，从另外一个线程的角度来看，它要么完成了，要么尚未发生，而不是进行中的一种状态。因此，原子性可以保证一个线程所读取到的共享变量的值要么是该变量的初始值要么是该变量的相对新值，而不是更新过程中的一个相当于“半成品”的值。设 Processor 0、Processor 1 和 Processor 2 上的 3 个线程按照如表 2-2 所示的线程交错顺序执行。

表 2-2 展示原子操作的一个示例线程交错顺序

处理器 时刻	Processor 0	Processor 1	Processor 2
t_1	a=1;	（执行其他操作）	（执行其他操作）
t_2	（执行其他操作）	a=2;	（执行其他操作）
t_3	（执行其他操作）	（执行其他操作）	b=a+1;

a 为 int 型共享变量，其初始值为 0

原子性可以保证在 t_3 时刻 Processor 2 上的线程读取到的共享变量 a 的值要么为 0，要么为 1 或者 2，但是它并不能保证该值会是 1 或者 2。相反，如果共享变量 a 的类型为 `long` 或者 `double`，此时由于 Java 语言规范不保证（但是 Java 虚拟机却可能保证）对这种类型变量写操作的原子性，因此 t_3 时刻 Processor 2 上的线程读取到的 a 值可能为 0、1、2，也可能是其他的看起来在代码中从来不存在的一个值！

可见性描述的是一个线程对共享变量的更新对于另外一个线程而言是否可见（或者说什么情况下可见）的问题。保障可见性意味着一个线程可以读取到相应共享变量的相对新值。例如，上述例子中保障可见性可以保证 t_3 时刻 Processor 2 上的线程读取到的共享变量 a 的值为 2（相对新值，这里恰好也是最新值）。

因此，从保障线程安全的角度来看，光保障原子性可能是不够的，有时候还要同时保障可见性。可见性和原子性同时得以保障才能够确保一个线程能够“正确”地看到（即看到的不是“半成品”）其他线程对共享变量所做的更新。

线程的启动、停止与可见性

Java 语言规范（JLS, Java Language Specification）保证，父线程在启动子线程之前对共享变量的更新对于子线程来说是可见的，如清单 2-8 所示。

清单 2-8 线程启动与可见性

```
public class ThreadStartVisibility {
    // 线程间的共享变量
    static int data = 0;

    public static void main(String[] args) {

        Thread thread = new Thread() {
            @Override
            public void run() {
                // 使当前线程休眠 R 毫秒（R 的值为随机数）
                Tools.randomPause(50);

                // 读取并打印变量 data 的值
                System.out.println(data);
            }
        };

        // 在子线程 thread 启动前更新变量 data 的值
        data = 1; // 语句①
        thread.start();
    }
}
```

```

// 使当前线程休眠 R 毫秒 (R 的值为随机数)
Tools.randomPause(50);

// 在子线程 thread 启动后更新变量 data 的值
data = 2; // 语句②
}
}

```

如果我们把上述程序中的语句②注释掉,则由于 main 线程在启动其子线程 thread 之前将共享变量 data 的值更新为 1(见语句①),因此子线程 thread 所读取到的共享变量 data 的值一定为 1。这是由于父线程在子线程启动前对共享变量的更新对子线程的可见性是有保证的;如果我们没有将语句②注释掉,那么由于父线程在子线程启动之后对共享变量的更新对子线程的可见性是没有保证的,因此子线程 thread 此时读取到的共享变量 data 的值可能为 2,也可能仍然为 1。这就解释了为什么多次运行上述程序可以发现其输出可能是“1”,也可能是“2”。

类似地,Java 语言规范保证一个线程终止后该线程对共享变量的更新对于调用该线程的 join 方法的线程而言是可见的,如清单 2-9 所示。

清单 2-9 线程终止与可见性

```

public class ThreadJoinVisibility {
    // 线程间的共享变量
    static int data = 0;

    public static void main(String[] args) {

        Thread thread = new Thread() {
            @Override
            public void run() {
                // 使当前线程休眠 R 毫秒 (R 的值为随机数)
                Tools.randomPause(50);

                // 更新 data 的值
                data = 1;
            }
        };

        thread.start();

        // 等待线程 thread 结束后, main 线程才继续运行
        try {
            thread.join();
        }
    }
}

```

```

    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    // 读取并打印变量 data 的值
    System.out.println(data);

}
}

```

在上述程序中，线程 `thread` 运行时将共享变量 `data` 的值更新为 1，因此 `main` 线程对线程 `thread` 的 `join` 方法调用结束后，该线程读取到的共享变量 `data` 值为 1 这一点是有保证的。

2.6 有序性

有序性（Ordering）指在什么情况下一个处理器上运行的一个线程所执行的内存访问操作在另外一个处理器上运行的其他线程看来是乱序的（Out of order）。所谓乱序，是指内存访问操作的顺序看起来像是发生了变化。在进一步介绍有序性这个概念之前，我们需要先介绍重排序的概念。为便于讨论，本书假定每个线程都是运行在各自的处理器上，即不考虑一个处理器上基于时间片（Time Slice）分时而实现的多线程（本章的扩展阅读部分会解释这里为何要做这个假设）。

2.6.1 重排序的概念

顺序结构是结构化编程中的一种基本结构，它表示我们希望某个操作必须先于另外一个操作得以执行。另外，两个操作即便是可以用任意一种顺序执行，但是反映在代码上这两个操作也总是有先后关系。但是在多核处理器的环境下，这种操作执行顺序可能是没有保障的：编译器可能改变两个操作的先后顺序；处理器可能不是完全依照程序的目标代码所指定的顺序执行指令；另外，一个处理器上执行的多个操作，从其他处理器的角度来看其顺序可能与目标代码所指定的顺序不一致。这种现象就叫作重排序（Reordering）。

重排序是对内存访问有关的操作（读和写）所做的一种优化，它可以在不影响单线程程序正确性的情况下提升程序的性能。但是，它可能对多线程程序的正确性产生影响，即它可能导致线程安全问题。与可见性问题类似，重排序也不是必然出现的。

重排序的潜在来源有许多，包括编译器（在 Java 平台中这基本上指 JIT 编译器）、处理器和存储子系统（包括写缓冲器 Store Buffer、高速缓存 Cache）。为了便于下面的讲解，

我们先定义几个与内存操作顺序有关的术语。

- 源代码顺序（Source Code）：源代码中所指定的内存访问操作顺序。
- 程序顺序（Program Order）：在给定的处理器上运行的目标代码（Object Code）所指定的内存访问操作顺序。尽管 Java 虚拟机执行 Java 代码有两种方式：解释执行（被执行的是字节码 Byte Code）和编译执行（被执行的是机器码）。为便于讨论，这里仅将目标代码定义为字节码。
- 执行顺序（Execution Order）：内存访问操作在给定的处理器上的实际执行顺序。
- 感知顺序（Perceived Order）：给定的处理器所感知到（看到）的该处理器及其他处理器的内存访问操作发生的顺序。

在此基础上，我们将重排序划分为指令重排序（Instruction Reorder）和存储子系统重排序两种，如表 2-3 所示。

表 2-3 重排序类型

重排序类型	重排序表现	重排序来源（主体）
指令重排序	程序顺序与源代码顺序不一致	编译器
	执行顺序与程序顺序不一致	JIT 编译器、处理器
存储子系统重排序	源代码顺序、程序顺序和执行顺序这三者保持一致，但是感知顺序与执行顺序不一致	高速缓存、写缓冲器

2.6.2 指令重排序

在源代码顺序与程序顺序不一致，或者程序顺序与执行顺序不一致的情况下，我们就说发生了指令重排序（Instruction Reorder）。指令重排序是一种动作，它确实确实地对指令的顺序做了调整，其重排序的对象是指令。

提示

Java 平台包含两种编译器：静态编译器（javac）和动态编译器（JIT 编译器）。前者的作用是将 Java 源代码（.java 文本文件）编译为字节码（.class 二进制文件），它是在代码编译阶段介入的。后者的作用是将字节码动态编译为 Java 虚拟机宿主机的本地代码（机器码），它是在 Java 程序运行过程中介入的。

在其他编译型语言（如 C++）中，编译器是可能导致指令重排序的：编译器出于性能的考虑，在其认为不影响程序（单线程程序）正确性的情况下可能会对源代码顺序进行调整，从而造成程序顺序与相应的源代码顺序不一致。在 Java 平台中，静态编译器（javac）

基本上不会执行指令重排序，而 JIT 编译器则可能执行指令重排序。下面我们通过一个 JIT 编译器指令重排序的实验（代码见清单 2-10）进一步了解指令重排序。

清单 2-10 JIT 编译器指令重排序 Demo

```
/**
 * 再现 JIT 指令重排序的 Demo
 *
 * @author Viscent Huang
 */
@ConcurrencyTest(iterations = 200000)
public class JITReorderingDemo {
    private int externalData = 1;
    private Helper helper;

    @Actor
    public void createHelper() {
        helper = new Helper(externalData);
    }

    @Observer({
        @Expect(desc = "Helper is null", expected = -1),
        @Expect(desc = "Helper is not null, but it is not initialized",
            expected = 0),
        @Expect(desc = "Only 1 field of Helper instance was initialized",
            expected = 1),
        @Expect(desc = "Only 2 fields of Helper instance were initialized",
            expected = 2),
        @Expect(desc = "Only 3 fields of Helper instance were initialized",
            expected = 3),
        @Expect(desc = "Helper instance was fully initialized", expected = 4) })
    public int consume() {
        int sum = 0;

        /**
         * 由于我们未对共享变量 helper 进行任何处理（比如采用 volatile 关键字修饰该变量），
         * 因此，这里可能存在可见性问题，即当前线程读取到的变量值可能为 null。
         */
        final Helper observedHelper = helper;
        if (null == observedHelper) {
            sum = -1;
        } else {
            sum = observedHelper.payloadA + observedHelper.payloadB
                + observedHelper.payloadC + observedHelper.payloadD;
        }

        return sum;
    }
}
```

```

    }

    static class Helper {
        int payloadA;
        int payloadB;
        int payloadC;
        int payloadD;

        public Helper(int externalData) {
            this.payloadA = externalData;
            this.payloadB = externalData;
            this.payloadC = externalData;
            this.payloadD = externalData;
        }
    }

    public static void main(String[] args) throws InstantiationException,
        IllegalAccessException {
        // 调用测试工具运行测试代码
        TestRunner.runTest(JITReorderingDemo.class);
    }
}

```

如清单 2-10 所示的程序非常简单（读者可以忽略其中的注解，因为那是给测试工具用的）：其中，`createHelper` 方法会将实例变量 `helper` 更新为一个新创建的 `Helper` 实例；`consume` 方法会读取 `helper` 所引用的 `Helper` 实例，并计算该实例的所有字段（`payloadA~payloadB`）的值之和作为其返回值。该程序的 `main` 方法调用测试工具 `TestRunner` 的 `runTest` 方法的作用是让测试工具安排一些线程并发地执行 `createHelper` 方法和 `consume` 方法，并统计 `consume` 方法多次执行的返回值。由于 `createHelper` 方法创建 `Helper` 实例的时候使用的构造器参数 `externalData` 值为 1，因此这样看来 `consume` 方法的返回值似乎“理所当然”地应该是 4。然而，事实却并不总是如此。使用如下命令以 `server` 模式并设置 Java 虚拟机参数 “`-XX:-UseCompressedOops`” 运行如清单 2-10 所示的程序⁶：

```
java -server -XX:-UseCompressedOops io.github.viscent.mtia.ch2.JITReorderingDemo
```

我们可以看到类似如下的输出⁷：

```
=====2016-03-13 15:05:40 Sun=====
```

6 虚拟机参数 “`-XX:-UseCompressedOops`”：<https://docs.oracle.com/javase/8/docs/technotes/guides/vm/performance-enhancements-7.html#compressedOop>。

7 这个输出相应的执行环境信息——操作系统为：Linux（x86_64 系统），JDK 版本为：JDK 1.8.0_40，处理器型号为：Intel i5-3210M。

```

expected:-1      occurrences:8      ==>Helper is null
expected:0      occurrences:2      ==>Helper is not null,but it is not initializ
ed
expected:1      occurrences:0      ==>Only 1 field of Helper instance was initia
lized
expected:2      occurrences:1      ==>Only 2 fields of Helper instance were init
ialized
expected:3      occurrences:4      ==>Only 3 fields of Helper instance were init
ialized
expected:4      occurrences:199985  ==>Helper instance was fully initialized

=====END=====

```

在上面的输出中, expected 后面的数字表示 consume 方法的返回值, 相应的 occurrences 表示出现相应返回值的次数。不难看出这次程序运行时, 有几次 consume 方法的返回值并不为 4: 有的为 3 (出现 4 次), 有的为 2 (出现 1 次), 甚至还有为 0 (出现 2 次)! 下面我们分析其中的原因。

createHelper 方法中的唯一一条语句:

```
helper = new Helper(externalData);
```

可以分解为以下几个子操作 (伪代码表示):

```

// 子操作①: 分配 Helper 实例所需的内存空间, 并获得一个指向该空间的引用
objRef = allocate(Helper.class);
// 子操作②: 调用 Helper 类的构造器初始化 objRef 引用指向的 Helper 实例
invokeConstructor(objRef);
// 子操作③: 将 Helper 实例引用 objRef 赋值给实例变量 helper
helper = objRef;

```

查看上述程序运行过程中 JIT 编译器动态生成的汇编代码 (相当于机器码), 如图 2-2 所示, 我们可以发现 JIT 编译器编译字节码的时候并不是每次都按照上述源代码顺序生成相应的机器码 (汇编代码): JIT 编译器将子操作③相应的指令重排到子操作②相应的指令之前, 即 JIT 编译器在初始化 Helper 实例之前可能已经将该实例的引用写入 helper 实例变量。这就导致了其他线程 (consume 方法的执行线程) 看到 helper 实例变量 (不为 null) 的时候, 该实例变量所引用的对象可能还没有被初始化或者未初始化完毕 (即相应构造器中的代码未执行结束)。这就解释了为什么我们在运行上述程序的时候, consume 方法的返回值有时候并不是 4。

```

0x00007f7c71148aed: mov     rsi,0x7f7c85522aa8 ; {metadata('io/github/viscent/mtia/ch2/JITReorderingDemo$Helper')}
0x00007f7c71148af7: call   0x00007f7c7106b8e0 ; *new; ==>子操作①:分配Helper实例所需的内存空间
0x00007f7c71148afc: mov     r11,rbp
0x00007f7c71148aff: mov     QWORD PTR [r11+0x18],rax ;*putfield helper ==>子操作②:将指向Helper实例的引用赋值给实例变量helper
0x00007f7c71148b03: mov     r10d,DWORD PTR [r11+0x10] ;*getfield externalData ==>读取实例变量externalData的值
0x00007f7c71148b07: mov     DWORD PTR [rax+0x10],r10d ;*putfield payloadA
0x00007f7c71148b0b: mov     DWORD PTR [rax+0x1c],r10d ;*putfield payloadD
0x00007f7c71148b0f: mov     DWORD PTR [rax+0x18],r10d ;*putfield payloadC
0x00007f7c71148b13: mov     DWORD PTR [rax+0x14],r10d ;*putfield payloadB

```

==>子操作③:对Helper实例进行初始化

图 2-2 JIT 编译器重排序 Demo 中的汇编代码片段

扩展阅读 如何查看 JIT 编译器动态生成的汇编代码？

查看 Java 虚拟机运行过程中动态生成的汇编代码，需要借助一款名为 hsdish 的反汇编插件（Disassembler plugin）。该插件可以从以下网址下载：

<https://kenai.com/projects/base-hsdis/downloads>

下面以 64 位的 Linux 系统下的 jdk1.8.0_40 环境为例进行讲解。

首先，下载文件 linux-hsdis-amd64.so。先将该文件下载到本机的任意目录，下载成功后将其重命名为 libhsdis-amd64.so。

接着，将 libhsdis-amd64.so 文件复制到 JDK 主目录下的 /jre/lib/amd64/server/子目录中。

接下来，我们就可以开始使用了。使用的时候只需要在启动 Java 程序的时候指定如下几个 Java 虚拟机扩展参数，如以下的启动命令：

```
java -server -XX:+UnlockDiagnosticVMOptions -XX:+PrintAssembly -XX:+LogCompilation
-XX:PrintAssemblyOptions=intel io.github.viscent.mtia.ch2.JITReorderingDemo
```

如果程序执行过程中打印出来的汇编代码比较多，直接在控制台上查看不方便，我们在 Java 虚拟机启动参数中再增加如下参数，就可以使汇编代码打印在日志文件中：

```
-XX:LogFile=/home/viscent/tmp/live.log
```

其中，/home/viscent/tmp/live.log 是日志文件的路径。

有关 JIT 编译器的更多扩展参数信息可以参考：

<https://docs.oracle.com/javase/8/docs/technotes/tools/unix/java.html#BABDDFII>

这个 Demo 明显地展示了重排序所具有的两个特征。

- 重排序可能导致线程安全问题：在本 Demo 中，重排序使得 consume 方法的返回值可能既不是 -1（表示 helper 实例为 null），也不是 4。当然，这并不表示重排序本身是错误的，而是说我们的程序本身有问题：我们的程序没有使用或者没有正确地使用线程同步机制（第 3 章会介绍这个概念）。
- 重排序不是必然出现的：本 Demo 运行时调用 consume 方法 200 000 次才出现 7（=4+1+2）次重排序（即 consume 方法返回值不为 -1，也不为 4），也就是出现重排序的比率是 0.035‰（=7/200000×1000‰）。

处理器也可能执行指令重排序，这使得执行顺序与程序顺序不一致。处理器对指令进行重排序也被称为处理器的乱序执行（Out-of-order Execution）。现代处理器为了提高指令执行效率，往往不是按照程序顺序逐一执行指令的，而是动态调整指令的顺序，做到哪条指令就绪就先执行哪条指令⁸，这就是处理器的乱序执行。在乱序执行的处理器中，指令是一条一条按照程序顺序被处理器读取的（亦即“顺序读取”），然后这些指令中哪条就绪了哪条就会先被执行，而不是完全按照程序顺序执行（亦即“乱序执行”）。这些指令执行的结果（要进行写寄存器或者写内存的操作）会被先存入重排序缓冲器（ROB，Reorder Buffer），而不是直接被写入寄存器或者主内存。重排序缓冲器会将各个指令的执行结果按照相应指令被处理器读取的顺序提交（Commit，即写入）到寄存器或者内存中去（亦即“顺序提交”）。在乱序执行的情况下，尽管指令的执行顺序可能没有完全依照程序顺序，但是由于指令的执行结果的提交（即反映到寄存器和内存中）仍然是按照程序顺序来的，因此处理器的指令重排序并不会对单线程程序的正确性产生影响。

处理器的乱序执行还采用了一种被称为猜测执行（Speculation）的技术。猜测执行技术就好比没有卫星导航时代在陌生地方开车遇到岔路口的情形：虽然我们不确定其中哪条路能够通往目的地，但是我们可以凭猜测（猜其能够到达目的地）走其中一条路，万一猜错了（前路不通）可以掉头重新走另外一条路。猜测执行能够造成 if 语句的语句体先于其条件语句被执行的效果，即可能导致指令重排序。例如，对于清单 2-11 中的语句③（if 语句）和语句⑤，处理器可能会先逐一读取数组 data 中的各个元素，并计算这些元素的和 sum，将其临时存放到 ROB 中，接着再去读取变量 ready 的值。如果 ready 的值为 true，那么再将 ROB 中临时存放的 sum 的值写到主内存中（通过高速缓存）。相反，如果 ready 的值为 false，那么通过丢弃 ROB 中临时存放的 sum 的值就可以实现该 if 语句的语句体（语句⑤）没有被执行到的效果。虽然，这并不会对单线程程序的正确性产生影响，但是它可能导致多线程程序出现非预期的结果：在 writer 方法和 reader 方法由不同的线程并发执行的情况下，语句⑤先于语句③被执行使得 data 数组的内容提前被读取，此时数组的内容可

8 例如一条指令所需的操作数都已经准备好。

能是其初始值 ([1, 2, 3, 4, 5, 6, 7, 8]), 而 reader 方法在源代码中先判断 ready 变量的值再读取 data 数组的目的是希望能够读取到 writer 方法执行线程更新后的数组内容 ([1, 1, 1, 1, 1, 1, 1, 1])。因此, 猜测执行可能使这个多线程程序出现了非预期的结果, 即影响了正确性。

清单 2-11 猜测执行示例代码

```
public class SpeculativeLoadExample {
    private boolean ready = false;
    private int[] data = new int[] { 1, 2, 3, 4, 5, 6, 7, 8 };

    public void writer() {
        int[] newData = new int[] { 1, 2, 3, 4, 5, 6, 7, 8 };
        for (int i = 0; i < newData.length; i++) {

            // 此处包含读内存的操作
            newData[i] = newData[i] - i;
        }
        data = newData; // 语句①
        // 此处包含写内存的操作
        ready = true; // 语句②
    }

    public int reader() {
        int sum = 0;
        int[] snapshot;
        if (ready) { // 语句③ (if 语句)
            snapshot = data;
            for (int i = 0; i < snapshot.length; i++) { // 语句④ (for 循环语句)
                sum += snapshot[i]; // 语句⑤
            }
        }
        return sum;
    }
}
```

可见, 处理器的指令重排序并不会对单线程程序的正确性产生影响, 但是它可能导致多线程程序出现非预期的结果。

2.6.3 存储子系统重排序

主内存 (RAM) 相对于处理器是一个慢速设备。为了避免其拖后腿, 处理器并不是直接访问主内存, 而是通过高速缓存 (Cache) 访问主内存的。在此基础上, 现代处理器还引入了写缓冲器 (Store Buffer, 也称 Write Buffer) 以提高写高速缓存操作 (以实现写

主内存）的效率。有的处理器（如 Intel 的 x86 处理器）对所有的写主内存的操作都是通过写缓冲器进行的。这里，我们将写缓冲器和高速缓存统称为存储子系统，它其实是处理器的子系统。

即使在处理器严格依照程序顺序执行两个内存访问操作的情况下⁹，在存储子系统的作用下其他处理器对这两个操作的感知顺序仍然可能与程序顺序不一致，即这两个操作的执行顺序看起来像是发生了变化。这种现象就是存储子系统重排序，也被称为内存重排序（Memory Ordering）。

指令重排序的重排序对象是指令，它实实在在地对指令的顺序进行调整，而存储子系统重排序是一种现象而不是一种动作，它并没有真正对指令执行顺序进行调整，而只是造成了一种指令的执行顺序像是被调整过一样的现象，其重排序的对象是内存操作的结果。习惯上为了便于讨论，在论及内存重排序问题的时候我们往往采用指令重排序的方式来表述，即我们也会用“内存操作 X 被重排序到内存操作 Y 之后”这样的表述称呼内存重排序。

约定 为了表述方便，在论及内存重排序的时候，本书仍然会采用指令重排序的方式来表述。例如，对于某个内存重排序，我们仍然会说“操作 Y 被重排序到操作 X 之前”之类的，读者需要注意这并非指针操作 Y 和操作 X 的指令重排序。

从处理器的角度来说，读内存操作的实质是从指定的 RAM 地址加载数据（通过高速缓存加载）到寄存器，因此读内存操作通常被称为 Load，写内存操作的实质是将数据（可能作为操作数直接存储在指令中，也可能存储在寄存器中）存储到指定地址表示的 RAM 存储单元中，因此写内存操作通常被称为 Store。所以，内存重排序实际上只有以下 4 种可能，如表 2-4 所示。

表 2-4 内存重排序

重排序类型	含 义
LoadLoad 重排序 (Loads reordered after loads)	该重排序指一个处理器上先后执行两个读内存操作 L1 和 L2，其他处理器对这两个内存操作的感知顺序可能是 L2→L1 ¹⁰ ，即 L1 被重排序到 L2 之后
StoreStore 重排序 (Stores reordered after stores)	该重排序指一个处理器上先后执行两个写内存操作 W1 和 W2，其他处理器对这两个内存操作的感知顺序可能是 W2→W1，即 W1 被重排序到 W2 之后

9 此处我们还假定源代码顺序、程序顺序和执行顺序这三者保持一致。

10 →符号表示箭头左侧的操作看起来先于箭头右侧的操作完成。

续表

重排序类型	含 义
LoadStore 重排序 (Loads reordered after stores)	该重排序指一个处理器上先后执行读内存操作 L1 和写内存操作 W2，其他处理器对这两个内存操作的感知顺序可能是 W2→L1，即 L1 被重排序到 W2 之后
StoreLoad 重排序 (Stores reordered after loads)	该重排序指一个处理器上先后执行写内存操作 W1 和读内存操作 L2，其他处理器对这两个内存操作的感知顺序可能是 L2→W1，即 W1 被重排序到 L2 之后

内存重排序与具体的处理器微架构有关，基于不同微架构的处理器所允许(或者支持) 的内存重排序是不同的，如表 2-5 所示。

表 2-5 不同处理器架构所支持的内存重排序

重排序类型	x86	x86 oostore	AMD64	IA-64	Alpha	ARMv7	zSeries
LoadLoad		Y		Y	Y	Y	
LoadStore		Y		Y	Y	Y	
StoreStore		Y		Y	Y	Y	
StoreLoad	Y	Y	Y	Y	Y	Y	Y

注：Y 表示处理器架构支持相应的重排序。资料来源：https://en.wikipedia.org/wiki/Memory_ordering。

内存重排序可能导致线程安全问题。假设处理器 Processor 0 和处理器 Processor 1 上的两个线程按照如表 2-6 所示的交错顺序各自执行其代码，其中 data、ready 是这两个线程的共享变量，其初始值分别为 0 和 false。Processor 0 上的线程所执行的处理逻辑是更新数据 data 并在此之后将相应的更新标志 ready 的值设为 true。Processor 1 上的线程所执行的处理逻辑是当数据更新标志 ready 的值不为 true（表示该线程所需的数据尚未被其他线程更新）时无限等待直到 ready 的值为 true，只有在 ready 的值为 true 的情况下才将 data 的值打印出来。

假设 Processor 0 依照程序顺序先后执行 S1 和 S2，那么 S1 和 S2 的操作结果会被先后写入写缓冲器中。但是由于某些处理器（比如 ARM 处理器）的写缓冲器为了提高将其中的内容写入高速缓存的效率而不保证写操作结果先入先出（FIFO，First-In-First-Out）的顺序，即较晚到达写缓冲器的写操作结果可能更早地被写入高速缓存，因此 S2 的操作结果可能先于 S1 的操作结果被写入高速缓存，即 S1 被重排序到 S2 之后（内存重排序）。这就导致了 Processor 1 上的线程读取到 ready 的值为 true 时，由于 S1 的操作结果仍然停留

在 Processor 0 的写缓冲器之中，而一个处理器并不能读取到另外一个处理器的写缓冲器中的内容，因此 Processor 1 上的线程读取到的 data 值仍然是其初始值 0。可见，此时内存重排序导致了 Processor 1 上的线程的处理逻辑无法达到其预期目标，即导致了线程安全问题。

表 2-6 StoreStore 重排序示例

Processor 0	Processor 1
data=1; //S1	
ready=true; //S2	
	while(!ready){;} //L3
	System.out.println(data); //L4

2.6.4 貌似串行语义

重排序并非随意地对指令、内存操作的结果进行杂乱无章的排序或者顺序调整，而是遵循一定的规则。编译器（主要是 JIT 编译器）、处理器（包括其存储子系统）都会遵守这些规则，从而给单线程程序创造一种假象（Illusion）——指令是按照源代码顺序执行的。这种假象就被称为貌似串行语义（As-if-serial Semantics）。貌似串行语义只是从单线程程序的角度保证重排序后的运行结果不影响程序的正确性，它并不保证多线程环境下程序的正确性，这点从如清单 2-10 所示程序的运行结果中不难看出。

为了保证貌似串行语义，存在数据依赖关系的语句不会被重排序，只有不存在数据依赖关系的语句才会被重排序。如果两个操作（指令）访问同一个变量（地址），且其中一个操作（指令）为写操作，那么这两个操作之间就存在数据依赖关系（Data Dependency），如表 2-7 所示。

表 2-7 数据依赖关系类型

类 型	代码示例	说 明
写后读（WAR）	x=1; y=x+1;	后一条语句的操作数包含前一条语句的执行结果
读后写（RAW）	y=x; x=1;	前一条语句读取一个变量后，后一条语句更新了该变量的值
写后写（WAW）	x=1; x=2;	两条语句对同一变量进行写操作

例如，下列代码中的语句③要进行的操作依赖于语句①和语句②执行后的结果数据，因此语句③是不允许和语句①或语句②进行重排序的；而语句①和语句②之间并没有数据依赖关系，它们可以以任意的顺序被执行（只要它们在语句③之前执行即可）。

```
float price = 59.0f; // 语句①
short quantity = 5; // 语句②
float subTotal = price * quantity; // 语句③
```

另外，存在控制依赖关系的语句是可以允许被重排序的。如果一条语句（指令）的执行结果会决定另外一条语句（指令）能否被执行，那么这两条语句（指令）之间就存在控制依赖关系（Control Dependency）。存在控制依赖关系的语句最典型的的就是 if 语句中的条件表达式和相应的语句体。允许这种重排序意味着处理器可能先执行 if 语句体所涉及的内存访问操作，然后再执行相应的条件判断（即 2.6.2 节介绍的猜测执行）！允许对存在控制依赖关系的语句进行重排序同样也是出于性能考虑。这是因为，存在控制依赖关系的语句（如 if 语句）会影响处理器对指令序列执行的并行程度。

扩展阅读 单处理器系统是否会受重排序的影响？

这个问题的答案是“会和不会都有可能”。读者可能还记得本章的开头我们所做的一个假设：假设每个线程都运行在各自的处理器上。之所以这么做，是因为重排序这个问题对于单处理器上实现的多线程而言有点特殊。我们知道，单处理器上实现的多线程实际上是通过分配时间片（Time Slice）实现的。单处理器的系统也存在重排序现象，那么单处理器上运行的一个线程上发生的重排序对这个处理器上运行的其他线程而言，其正确性是否会受到影响呢？

编译期重排序，即静态编译器（对于 Java 平台指 javac）造成的重排序会对运行在单处理器上的多个线程产生影响。例如，如下的源代码：

```
data=1; // 语句①
ready=true; // 语句②
```

假设编译器在编译的时候将代码的顺序调整为：

```
ready=true; // 语句②
data=1; // 语句①
```

在这种情况下，当一个线程运行到语句②的时候，此时如果发生了上下文切换（这个概念 2.7 节会介绍），另外一个线程被切换进来运行，那么这个被切入的线程看到 ready 值为 true 的时候，实际上语句①还没有被切出的线程执行。因此，此时的重排序就可能影响该切入线程的正确性。

运行期重排序，包括存储子系统造成的重排序、JIT 编译器造成的重排序以及处理器的乱序执行所导致的重排序，并不会对单处理器上运行的多线程产生影响，即在这些线程看来处理器像是按照程序顺序执行指令。这是因为，这些重排序都是运行期实现的，即当

这些重排序发生的时候，相关指令还没有完全执行完毕，即它们的执行结果还没有被提交到主内存，此时处理器（在系统只有一个处理器的情况下）通常不会立即进行上下文切换以运行另外一个线程，而是等这些正在执行的指令执行完毕之后再进行上下文切换。也就是说，当前线程被切出而另外一个线程被切入的时候，这个被切出的线程被重排序的操作（指令）已经执行完毕了，因此重排序对于这个被切入的线程而言就像是不存在一样。

2.6.5 保证内存访问的顺序性

我们知道硬件和软件的因素都可能导致程序的感知顺序与源代码顺序不一致，而这种不一致可能导致线程安全问题。那么，如何避免重排序导致的线程安全问题呢？这个问题实质上就是如何保证感知顺序与源代码顺序一致，即有序性。

我们知道貌似串行语义只是保障重排序不影响单线程程序的正确性。从这个角度出发，有序性的保障可以理解为通过某些措施使得貌似串行语义扩展到多线程程序，即重排序要么不发生，要么即使发生了也不会影响多线程程序的正确性。因此，有序性的保障也可以理解为从逻辑上部分禁止重排序。当然，这并不意味着从物理上禁止重排序而使得处理器完全依照源代码顺序执行指令，因为那样性能太低！因此，本书后续提到的禁止重排序，都是指逻辑上的部分禁止重排序。

从底层的角度来说，禁止重排序是通过调用处理器提供相应的指令（内存屏障）来实现的¹¹。当然，Java 作为一个跨平台的语言，它会替我们与这类指令打交道，而我们只需要使用语言本身提供的机制即可。前面我们介绍的 `volatile` 关键字、`synchronized` 关键字都能够实现有序性。例如，如清单 2-11 所示的程序为了实现有序性，可以将其对 `ready` 变量的声明由

```
private static boolean ready = false;
```

修改为

```
private static volatile boolean ready = false;
```

这里，`volatile` 关键字的使用禁止了清单 2-11 中的语句①和语句②的重排序。

有关 `volatile` 关键字、`synchronized` 关键字以及重排序，在第 3 章我们会进一步讲解。

11 这些指令被称为内存屏障 Memory Barrier，它们是因具体的处理器而异的，如 Intel 系列提供的是 `sfence`、`lfence` 和 `mfence` 指令。

扩展阅读 可见性与有序性的联系与区别

可见性是有序性的基础。可见性描述的是一个线程对共享变量的更新对于另外一个线程是否可见，或者说什么情况下可见的问题。有序性描述的是，一个处理器上运行的线程对共享变量所做的更新，在其他处理器上运行的其他线程看来，这些线程是以什么样的顺序观察到这些更新的问题。因此，可见性是有序性的基础。另一方面，二者又是相互区分的。

有序性影响可见性。由于重排序的作用，一个线程对共享变量的更新对于另外一个线程而言可能变得不可见。例如，在如清单 2-8 所示的程序中，假如 JIT 编译器将语句①重排序到 `thread.start()` 之后执行，那么父线程对共享变量 `data` 的更新对于子线程 `thread` 而言的可见性就无法保证了。当然，事实上 Java 语言规范不允许像语句①这样的写操作（这种操作被称为普通写，Normal Store）与 `Thread.start()` 语句进行重排序（包括指令重排序和内存重排序）。

2.7 上下文切换

上下文切换（Context Switch）在某种程度上可以被看作多个线程共享同一个处理器的产物¹²，它是多线程编程中的一个重要概念。

2.7.1 上下文切换及其产生原因

我们知道，在单处理器（Uni-processor）上也能够以多线程的方式实现并发，即一个处理器可以在同一时间段内运行多个线程。这好比即使是一次只能够被一个儿童操纵的玩具（如遥控四驱车）也能够被多个儿童一起玩的情形：每个儿童玩一定的时间，时间到了，这个玩具必须交给另外一个儿童玩，依此规则各个儿童轮流玩。这里，每个儿童可以占用玩具进行玩耍的时间就被称为时间片（Time Slice）。单处理器上的多线程其实就是通过这种时间片分配的方式实现的。时间片决定了一个线程可以连续占用处理器运行的时间长度。当一个进程中的一个线程由于其时间片用完或者其自身的原因（比如，它需要稍后再继续运行）被迫或者主动暂停其运行时，另外一个线程（可能是同一个进程或者其他进程中的一个线程）可以被操作系统（线程调度器）选中占用处理器开始或者继续其运行。这种一个线程被暂停，即被剥夺处理器的使用权，另外一个线程被选中开始或者继续运行的过程就叫作线程上下文切换。为了方便，本书也将线程上下文切换简单地称为上下文切换¹³。

¹² 本书所讲的上下文切换具体指线程上下文切换，我们不讨论进程上下文切换。

¹³ 这表示我们不考虑进程上下文切换，尽管它是一个与线程上下文切换比较相似的概念。

相应地，一个线程被剥夺处理器的使用权而被暂停运行就被称为切出（Switch Out）；一个线程被操作系统选中占用处理器开始或者继续其运行就被称为切入（Switch In）。

可见，我们看着是连续运行的线程，实际上是以断断续续运行的方式使其任务进展的。这种方式意味着在切出和切入的时候操作系统需要保存和恢复相应线程的进度信息，即切入和切出那一刻相应线程所执行的任务进行到什么程度了（如计算的中间结果以及执行到了哪条指令）。这个进度信息就被称为上下文（Context）。它一般包括通用寄存器（General Purpose Register）的内容和程序计数器（Program Counter）的内容。在切出时，操作系统需要将上下文保存到内存中，以便被切出的线程稍后占用处理器继续其运行时能够在此基础上进展。在切入时，操作系统需要从内存中加载（恢复）被选中线程的上下文，以在之前运行的基础上继续进展。

上下文切换类似于我们接听手机电话的场景：我们正在接听一个电话并与对方讨论某件事情的时候突然有另外一个来电，此时通常会跟对方说“我先接个电话，你别挂断”，并记下与他的讨论进行到什么程度了（保存上下文）。然后，接听新的来电并告诉对方稍后会对其电话回拨并将该来电挂断。接着，我们又继续先前的讨论（恢复上下文）。如果在接听新来电之前，我们没有特意记下当前的讨论进展到什么程度了（即上下文），等我们接听新的来电后再回过头继续讨论时，可能得问对方“刚才我们讲到哪里了”这样的问题。

从 Java 应用的角度来看，一个线程的生命周期状态在 RUNNABLE 状态与非 RUNNABLE 状态（包括 BLOCKED、WAITING 和 TIMED_WAITING 中的任意一个子状态）之间切换的过程就是一个上下文切换的过程。当一个线程的生命周期状态由 RUNNABLE 转换为非 RUNNABLE 时，我们称这个线程被暂停。线程的暂停就是相应线程被切出的过程，这里操作系统会保存相应线程的上下文，以便该线程稍后再次进入 RUNNABLE 状态时能够在之前执行进度的基础上进展。而一个线程的生命周期状态由非 RUNNABLE 状态进入 RUNNABLE 状态时，我们就称这个线程被唤醒（Wakeup）。一个线程被唤醒仅代表该线程获得了一个继续运行的机会，而并不代表其立刻可以占用处理器运行。因此，当被唤醒的线程被操作系统选中占用处理器继续其运行的时候，操作系统会恢复之前为该线程保存的上下文，以便其在此基础上进展。

2.7.2 上下文切换的分类及具体诱因

按照导致上下文切换的因素划分，我们可以将上下文切换分为自发性上下文切换（Voluntary Context Switch）和非自发性上下文切换（Involuntary Context Switch）。

自发性上下文切换指线程由于其自身因素导致的切出。从 Java 平台的角度来看，一

个线程在其运行过程中执行下列任意一个方法都会引起自发性上下文切换。

- `Thread.sleep(long millis)`
- `Object.wait()/wait(long timeout)/wait(long timeout, int nanos)`
- `Thread.yield()`¹⁴
- `Thread.join()/Thread.join(long timeout)`
- `LockSupport.park()`

另外，线程发起了 I/O 操作（如读取文件）或者等待其他线程持有的锁（锁的概念在第3章会介绍）也会导致自发性上下文切换¹⁵。

非自发性上下文切换指线程由于线程调度器的原因被迫切出。导致非自发性上下文切换的常见因素包括被切出线程的时间片用完或者有一个比被切出线程优先级更高的线程需要被运行。从 Java 平台的角度来看，Java 虚拟机的垃圾回收（Garbage Collect）动作也可能导致非自发性上下文切换。这是因为垃圾回收器在执行垃圾回收的过程中可能需要暂停所有应用线程（Stop-the-world）才能完成其工作，比如在主要回收（Major Collection）过程中，垃圾回收器在对 Java 虚拟机堆内存区域进行整理（Compact）的时候需要先停止所有应用线程。

2.7.3 上下文切换的开销和测量

一方面，上下文切换是必要的。即使是在多核处理器系统中上下文切换也是必要的，这是因为一个系统上需要运行的线程的数量相对于这个系统所拥有的处理器数量总是要大得多（“僧多粥少”）。另一方面，上下文切换又有其不容小觑的开销。

从定性的角度来说，上下文切换的开销包括直接开销和间接开销。其中，直接开销包括：

- 操作系统保存和恢复上下文所需的开销，这主要是处理器时间开销。
- 线程调度器进行线程调度的开销（比如，按照一定的规则决定哪个线程会占用处理器运行）。

14 `Thread.yield()`调用可能会也可能不会导致上下文切换，这具体取决于线程调度器。

15 确切地说是阻塞式（Blocking）I/O 会导致上下文切换。

间接开销包括：

- 处理器高速缓存重新加载的开销。一个被切出的线程可能稍后在另外一个处理器上被切入继续运行。由于这个处理器之前可能未运行过该线程，那么这个线程在其继续运行过程中需访问的变量仍然需要被该处理器重新从主内存或者通过缓存一致性协议从其他处理器加载到高速缓存之中。这是有一定时间消耗的。
- 上下文切换也可能导致整个一级高速缓存中的内容被冲刷（Flush），即一级高速缓存中的内容会被写入下一级高速缓存（如二级高速缓存）或者主内存（RAM）中。

从定量的角度来说，一次上下文切换的时间消耗是微秒（ μs ）级的¹⁶。

第1章我们提到的“和尚挑水”的故事说明线程的数量越多，它们可能导致的上下文切换的开销也就可能越大。因此，多线程编程中使用的线程数量越多，程序的计算效率可能反而越低！因此，在设计多线程程序的时候，减少上下文切换也是一个重要的考量因素。

那么，我们如何测量一个多线程程序在上下文切换上面的具体开销呢？这个问题可以转换为另外一个问题来回答：如何确定一个多线程程序在某个时间段或者某种场景下运行时发生的上下文切换（主要是自发性上下文切换）的次数。

在 Linux 平台下，我们可以使用 Linux 内核提供的 `perf` 命令来监视 Java 程序运行过程中的上下文切换的次数和频率。例如，我们可以使用 `perf` 命令来监视如清单 1-9 所示的程序运行，相应的命令如下：

```
perf stat -e cpu-clock,task-clock,cs,cache-references,cache-misses java io.github.
viscent.mtia.ch1.FileDownloaderApp http://server.com/a.png http://server.net/b.
png http://server.info/c.png
```

在上述命令中，参数 `e` 的值中的 `cs` 表示被监视程序的上下文切换的数量。上述命令执行后输出的内容类似如下：

```
185.957629 cpu-clock (msec)
185.968432 task-clock (msec)      #    0.034 CPUs utilized
653 cs                          #    0.004 M/sec
5,032,903 cache-references        #   27.063 M/sec
900,864 cache-misses              #   17.899 % of all cache refs

5.513332129 seconds time elapsed
```

¹⁶ $1000\mu\text{s} = 1\text{ms}$

由此可见，如清单 1-9 所示的程序的这次运行一共产生了 653 次上下文切换（包括自发性上下文切换和非自发性上下文切换）。

在 Windows 平台下，我们可以使用 Windows 自带的工具 `perfmon` 来监视 Java 程序运行过程中的上下文切换情况¹⁷。

多线程编程相比于单线程编程来说，它意味着更多的上下文切换。因此，多线程编程不一定就比单线程编程的计算效率更高。

2.8 线程的活性故障

线程是为任务而生的。因此，理想情况下我们希望线程一直处于 `RUNNABLE` 状态。显然，事实并非如此：导致一个线程可能处于非 `RUNNABLE` 状态的因素除了资源（主要是处理器资源有限而导致的上下文切换）限制之外，还有程序自身的错误和缺陷。这些由资源稀缺性或者程序自身的问题和缺陷导致线程一直处于非 `RUNNABLE` 状态，或者线程虽然处于 `RUNNABLE` 状态但是其要执行的任务却一直无法进展的现象就被称为线程活性故障（Liveness Failure）。

常见的活性故障包括以下几种。

- 死锁（Deadlock）。死锁好比鹬蚌相争故事中的情形：鹬啄住蚌的肉，蚌夹住鹬的嘴。鹬对蚌说：“你先放开我的嘴我就不啄你的肉。”而蚌对鹬说：“你先放开我的肉我就不夹你的嘴。”于是最后谁也不放开谁！死锁产生的典型场景是一个线程 X 持有资源 A 的时候等待另外一个线程释放资源 B，而另外一个线程 Y 在持有资源 B 的时候却等待线程 X 释放资源 A。死锁的外在表现是当事线程的生命周期状态永远处于非 `RUNNABLE` 状态而使其任务一直无法进展。
- 锁死（Lockout）。锁死就好比睡美人的故事中睡美人醒来的前提是她要得到王子的亲吻，但是如果王子无法亲吻她（比如王子“挂了”……），那么睡美人将一直沉睡！
- 活锁（Livelock）。活锁好比小猫试图咬自己的尾巴，虽然它总是追着自己的尾巴咬，但却始终无法咬到。活锁的外在表现是线程可能处于 `RUNNABLE` 状态，但是线程所要执行的任务却丝毫没有进展，即线程可能一直在做无用功。
- 饥饿（Starvation）。饥饿好比母鸟给雏鸟喂食的情形，健壮的雏鸟总是抢先从母鸟的嘴里抢到食物，从而导致那些弱小的雏鸟总是挨饿。饥饿就是线程因无法获

¹⁷ `perfmon` 的可执行文件为：Windows 安装目录\System32\perfmon.exe。

得其所需的资源而使得任务执行无法进展的现象。

第 7 章我们会进一步介绍活性故障。

2.9 资源争用与调度

由于资源的稀缺性（就像处理器资源那样“僧多粥少”）或者资源本身的特性（例如打印机一次只能打印一个文件），我们往往需要在多个线程间共享同一个资源。一次只能被一个线程占用的资源被称为排他性（Exclusive）资源。常见的排他性资源包括处理器、数据库连接、文件等。在一个线程占用一个排他性资源进行访问（读、写操作）而未释放其对资源所有权的时候，其他线程试图访问该资源的现象就被称为资源争用（Resource Contention），简称争用。显然，争用是在并发环境下产生的一种现象。同时试图访问同一个已经被其他线程占用的资源的线程数量越多，争用的程度就越高；反之争用的程度就越低。相应的争用就被分别称为高争用和低争用。

约定

同一时间内，处于运行状态（即生命周期状态为 RUNNABLE 的 RUNNING 子状态的线程）的线程数量越多，我们就称并发的程度越高，简称高并发，如图 2-3（a）所示。高并发是相对于低并发（如图 2-3（b）所示）而言的。图 2-3 中的每个线条代表的是相应线程的运行情况，其中实线条表示相应时间段内的相应线程处于 RUNNING 状态，虚线条表示相应时间段内的相应线程处于非 RUNNABLE 状态。

虽然高并发增加了争用的概率，但是高并发未必就意味着高争用。这好比车辆通过收费站的情形：在收费站仅支持人工收费（现金或者刷卡）的情况下，虽然车辆（线程）在其到达收费站之前行驶在各自的车道上（并发），但是过收费站的那一刻车辆可能需要等待前面的车辆缴费完毕放行之后它才能通行，因此此时争用（过收费站的通道）的概率较大。在多数或所有车辆使用 ETC（Electronic Toll Collection，电子不停车收费系统）付费的情况下，尽管车流量很大（高并发），但是由于多数车辆都可以直接通过收费站而无须停车手工缴费，因此这些车辆基本上不需要特意等待前面的车辆通过收费站（低争用）。可见，我们要达到的理想情况是高并发、低争用。但是理想总归是理想，这当中是存在不少阻碍的。

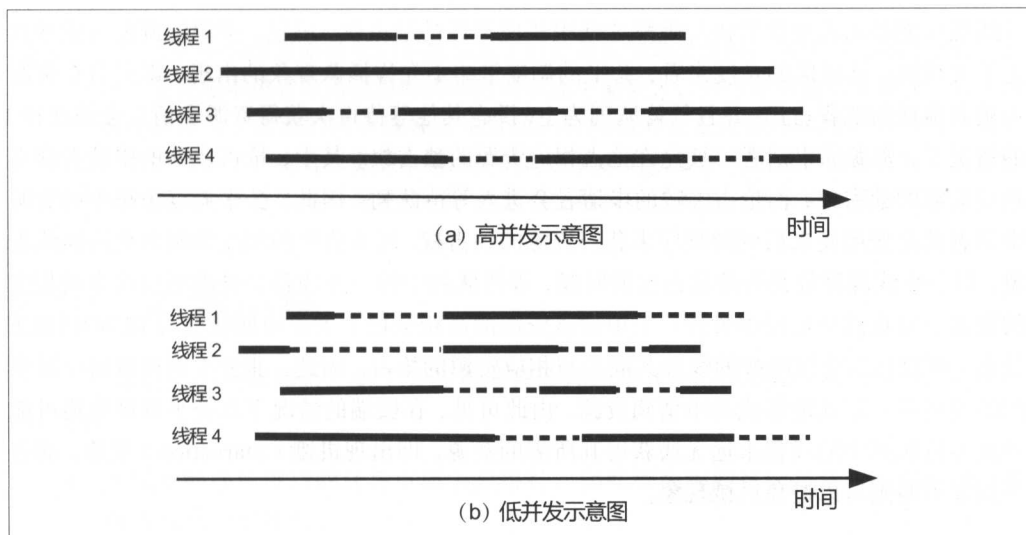


图 2-3 高并发示意图

而多个线程共享同一个资源又会带来新的问题，即资源的调度问题。在多个线程申请同一个排他性资源的情况下，决定哪个线程会被授予该资源的独占权，即选择哪个申请者占用该资源的过程就是资源的调度。获得资源的独占权而又未释放其独占权的线程就被称为该资源的持有线程。资源调度策略的一个常见特性就是它能否保证公平性。所谓公平性 (Fairness)，是指资源的申请者 (线程) 是否按照其申请 (请求) 资源的顺序而被授予资源的独占权。如果资源的任何一个先申请者总是能够比任何一个后申请者先获得该资源的独占权，那么相应的资源调度策略就被称为是公平的 (Fair)；如果资源的后申请者可能比先申请者先获得该资源的独占权，那么相应的资源调度策略就被称为是非公平的 (Non-fair)。需要注意的是，不公平的资源调度策略往往只是说明它并不保证资源调度的公平性，即它允许不公平的资源调度的出现，而不是表示它刻意造就不公平 (Unfair) 的资源调度。

提示

以下内容可以在阅读本书后续章节遇到疑惑时再回头来看。

资源调度的一种常见策略就是排队。资源调度器 (负责资源调度) 内部维护一个等待队列，在存在资源争用的情况下，申请失败 (即没有获得资源的独占权) 的资源申请者 (线程) 会被存入该队列。通常，被存入等待队列的线程会被暂停。当相应的资源被其持有线程释放时，等待队列中的一个线程会被选中并被唤醒而获得再次申请资源的机会。被唤醒的线程如果申请到资源的独占权，那么该线程会从等待队列中移除；否则，该线程仍然会停留在等待队列中等待再次申请的机会，即该线程会再次被暂停。因此，等待队列中的等

待线程可能经历若干次暂停与唤醒才获得相应资源的独占权。可见，资源的调度可能导致上下文切换。从排队的角度来看，公平的调度策略不允许插队现象的出现，即只有在资源未被其他任何线程占用，并且等待队列为空（没有其他等待再次获得资源申请机会的线程）的情况下，资源的申请者才被允许抢占相应资源的独占权。其中，抢占成功的申请者获得相应资源的独占权，而抢占失败的申请者会进入等待队列。因此，公平调度策略中的资源申请者总是按照先来后到的顺序来获得资源的独占权。而非公平的调度策略则允许插队现象，即一个线程释放其资源独占权的时候，等待队列中的一个线程会被唤醒再次申请相应的资源，而在这个过程中另外一个申请该资源的活跃线程（生命周期状态为 `RUNNABLE` 状态）可以与这个被唤醒的线程共同参与相应资源的抢占。因此，非公平调度策略中被唤醒的线程不一定就能够成功申请到资源。由此可见，在极端的情况下非公平调度策略可能导致等待队列中的线程永远无法获得其所需的资源，即出现饥饿（`Starvation`）现象，而公平调度策略则可以避免饥饿现象。

一般来说，非公平调度策略的吞吐率较高，即单位时间内它可以为更多的申请者调配资源。其缺点是，从申请者个体的角度来看这些申请者获得相应资源的独占权所需时间的偏差可能比较大，即有的线程很快就申请到资源而有的线程则要经历若干次暂停与唤醒才成功申请到资源。公平调度策略的吞吐率较低，这是其维护资源独占权的授予顺序的开销比较大（主要是线程的暂停与唤醒所导致的上下文切换）的结果。其优点是，从申请者个体的角度来看这些申请者获得相应资源的独占权所需时间的偏差可能比较小，即每个申请者成功申请到资源所需的时间基本相同。在非公平调度策略中，资源的持有线程释放该资源的时候等待队列中的一个线程会被唤醒，而该线程从被唤醒到其继续运行可能需要一段时间。在该时间内，新来的线程（活跃线程）可以先被授予该资源的独占权。如果这个新来的线程占用该资源的时间不长，那么它完全有可能在被唤醒的线程继续其运行前释放相应的资源，从而不影响该被唤醒的线程申请资源。这种情形下，非公平调度策略带来一个好处——它可能减少上下文切换的次数（例如，前面例子中新来的线程无须被暂停和唤醒就申请到资源）。相反，如果多数（或者每个）线程占用资源的时间相当长，那么允许新来的线程抢先占用被释放的资源丝毫不会带来任何好处，反而会导致被唤醒的线程需要再次经历暂停和唤醒，从而增加了上下文切换。因此，多数（或者每个）线程占用资源的时间相当长（或者申请资源的间隔相对长）的情况下不适合使用非公平调度策略。因此，在没有特别需要的情况下，我们默认选择非公平调度策略即可。在资源的持有线程占用资源的时间相对长或线程申请资源的平均间隔时间相对长的情况下，或者对资源申请所需的时间偏差有所要求（即时间偏差较小）的情况下，可以考虑使用公平调度策略。

提示

非公平调度策略是我们多数情况下的首选资源调度策略。其优点是吞吐率较大；缺点是资源申请者申请资源所需的时间偏差可能较大，并可能导致饥饿现象。公平调度策略适合在资源的持有线程占用资源的时间相对长或资源的平均申请时间间隔相对长的情况下，或者对资源申请所需的时间偏差有所要求的情况下使用。其优点是线程申请资源所需的时间偏差较小，并且不会导致饥饿现象；其缺点是吞吐率较小。

2.10 本章小结

本章通过一些具体概念介绍了多线程编程的目标及其面临的挑战。本章知识结构如图2-4所示。

- 单线程程序所进行的计算本质上是串行。多线程编程的目标是将原本串行的计算改为并发乃至并行。
- 竞态 (Race Condition) 是指计算的正确性依赖于相对时间顺序 (Relative Timing) 或者线程的交错 (Interleaving)。竞态表现为计算的结果时而正确时而错误，它并不意味着计算的结果一定是错误的，其往往伴随着读取脏数据、丢失更新的问题。竞态是访问 (读取、更新) 同一组共享变量的多个线程所执行的操作相互交错 (Interleave) 而导致的干扰 (读取脏数据) 或者冲突 (丢失更新) 的结果。二维表分析法是分析和解释竞态的有效和常用工具。一个类能够导致竞态，那么它就不是线程安全的。线程安全意味着不存在竞态，但是不存在竞态却未必意味着线程安全。
- 线程安全问题表现为原子性、可见性和有序性这三个方面。这几个方面既相互区别，又相互联系。原子性的保障能够消除竞态。可见性描述了一个线程对共享变量的更新对于另外一个线程而言是否可见，或者说什么情况下可见的问题。原子性和可见性一同得以保障了一个线程能够共享变量的相对新值，而不是一个“半成品”的值。有序性描述了一个处理器上运行的一个线程对共享变量所做的更新，在另外一个处理器上运行的其他线程看来，这些线程是以什么样的顺序观察到这些更新的问题。可见性是有序性的基础，而有序性又可能影响可见性。
- 原子操作是“不可分割”的操作。所谓“不可分割”包括两层含义：其一，访问 (读、写) 某个共享变量的操作从其执行线程以外的任何线程来看，该操作要么已经执行结束，要么尚未发生，即其他线程不会“看到”该操作执行了部分的中间效果；其二，访问同一组共享变量的原子操作是不能够被交错的，这通常意味着互斥 (Mutual Exclusion)，即对于访问同一组共享变量的多个原子操作，一个线程执行其中一个操作的时候其他线程无法访问这组共享变量中的任意一个变量。将 read-modify-write 操作和 check-then-act 转换为原子操作能够消除竞态。在 Java

语言中，对 `long/double` 型以外的任何变量的写操作都是原子的。`volatile` 关键字修饰的 `long/double` 型写操作也具有原子性。针对任何变量的读操作都是原子操作。

- 可见性问题不是必然出现的，而一旦出现则可能导致灾难性后果。导致可见性问题的因素既有软件因素（JIT 编译器）也有硬件因素（处理器和内存等存储设备）。可见性的保障仅仅意味着一个线程能够读取到共享变量的相对新值，而不能保障该线程能够读取到相应变量的最新值。父线程在启动子线程前对共享变量所做的更新对这个子线程可见，子线程执行期间对共享变量所做的更新对该线程的 `join()` 执行线程可见（从 `join()` 返回处开始才是可见的）。
- 编译器、处理器、存储子系统（写缓冲器和高速缓存等）和运行时（JIT 编译器）都可能导致重排序。重排序是出于性能的需要并在满足“貌似串行语义”的前提下进行的，它可能导致线程安全问题。与可见性问题类似，重排序也不是必然出现的。有序性的保障是通过部分地从逻辑上禁止重排序实现的。可见性是有序性的基础，而有序性反过来又可能影响可见性。
- 上下文切换可以被看作多线程编程的必然产物，一方面它使得充分利用极其有限的处理器资源成为可能；另一方面它也增加了系统的开销。因此，多线程编程未必比单线程的计算效率要高。程序运行过程中发生的上下文切换既有自发性上下文切换，也有非自发性上下文切换。Linux 内核提供的 `perf` 命令可以帮助我们测量程序运行过程中发生的上下文切换的次数和频率。
- 多线程程序可能由于资源稀缺性或者程序自身的错误和缺陷而一直处于非 `RUNNABLE` 状态，或者即使是处于 `RUNNABLE` 状态，但是其要执行的任务一直无法进展，即产生了活性故障。
- 非公平调度策略是我们多数情况下的首选资源调度策略。其优点是吞吐率较大；缺点是资源申请者申请资源所需的时间偏差可能较大，并可能导致饥饿现象。公平调度策略适合在资源的持有线程占用资源的时间相对长或资源的平均申请时间间隔相对长的情况下，或者对资源申请所需的时间偏差有所要求的情况下使用。其优点是线程申请资源所需的时间偏差较小，并且不会导致饥饿现象；其缺点是吞吐率较小。

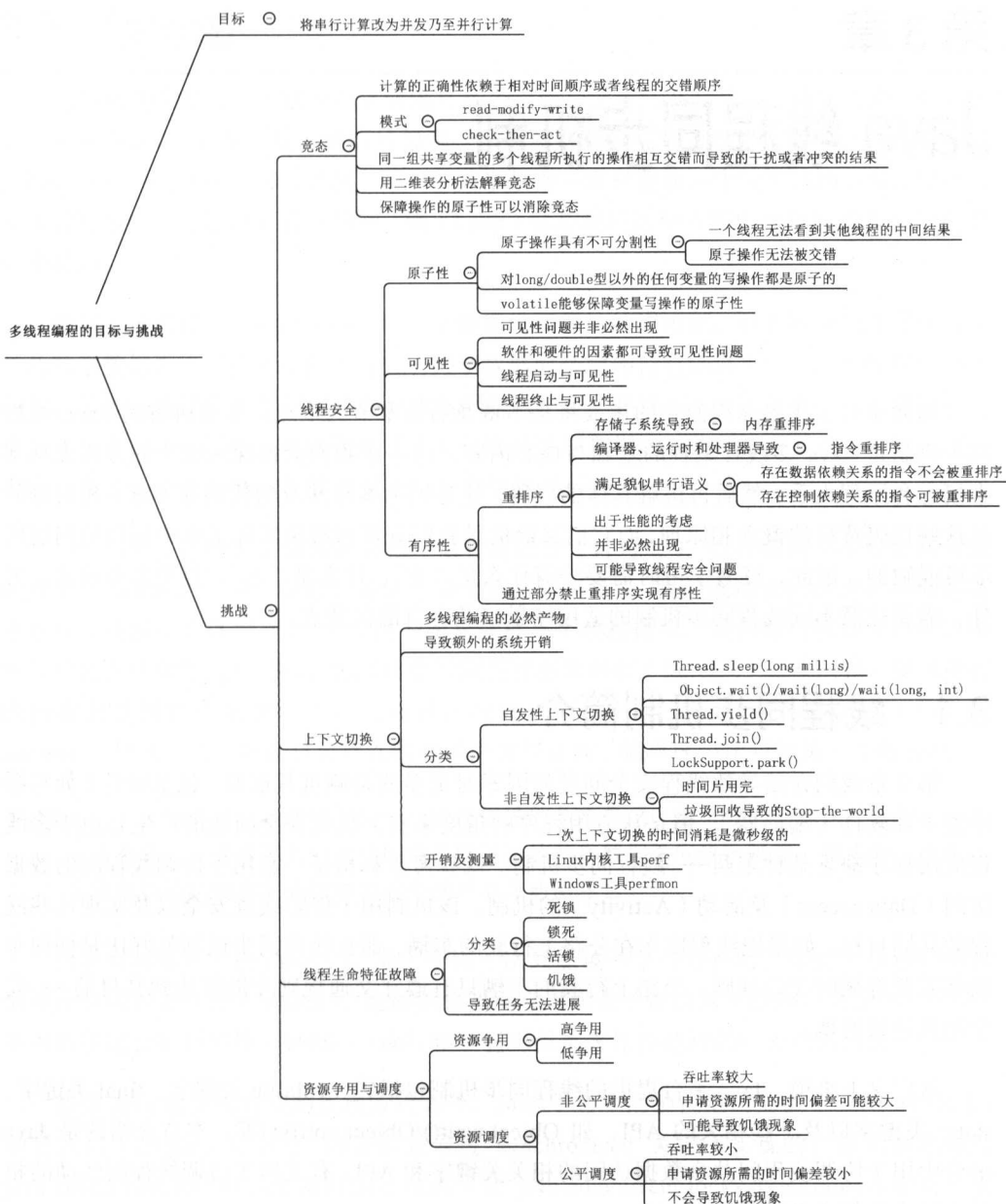


图 2-4 本章知识结构图

第3章

Java 线程同步机制

如何应对多线程编程面临的挑战将是本章及后续章节的重点。本章讲解的 Java 线程同步机制是 Java 多线程编程的基础与核心内容。这一章我们会以深入浅出的方式先从基本概念和原理入手，然后再讲解具体的代码。这是因为本章涉及的代码并不难，相对难的是这些代码背后的概念和原理，而它们又恰恰是我们分析和解决实际工作中遇到的问题所不可或缺的。因此，读者学习时需要分清什么是“本”，什么是“末”，以免舍本逐末。另外，需要注意不同线程同步机制的适用场景以及各自的优缺点。

3.1 线程同步机制简介

第2章我们介绍导致线程安全问题的因素时更多的是侧重其根源，包括硬件（如写缓冲器）和软件（编译器）。但是从应用程序的角度来看，线程安全问题的产生是由于多线程应用程序缺乏某种东西——线程同步机制。线程同步机制是一套用于协调线程间的数据访问（Data access）及活动（Activity）的机制，该机制用于保障线程安全以及实现这些线程的共同目标。如果把线程比作在公路上行驶的车辆，那么线程同步机制就好比是任何车辆都需要遵循的交通规则。公路上行驶的车辆只有遵守交通规则才能够达到其目的——安全地到达目的地。

从广义上来说，Java 平台提供的线程同步机制包括锁、volatile 关键字、final 关键字、static 关键字以及一些相关的 API，如 Object.wait()/Object.notify()等。本章介绍的是 Java 平台中用于协调线程间共享数据访问的相关关键字和 API。有关用于协调线程间活动的相关 API 会在第5章介绍。

3.2 锁概述

我们知道线程安全问题的产生前提是多个线程并发访问共享变量、共享资源（以下统称为共享数据）。于是，我们很容易想到一种保障线程安全的方法——将多个线程对共享数据的并发访问转换为串行访问，即一个共享数据一次只能被一个线程访问，该线程访问结束后其他线程才能对其进行访问。锁（Lock）就是利用这种思路以保障线程安全的线程同步机制。

按照上述思路，锁可以理解为对共享数据进行保护的许可证。对于同一个许可证所保护的共享数据而言，任何线程访问这些共享数据前必须先持有该许可证。一个线程只有在持有许可证的情况下才能够对这些共享数据进行访问；并且，一个许可证一次只能被一个线程持有；许可证的持有线程在其结束对这些共享数据的访问后必须让出（释放）其持有的许可证，以便其他线程能够对这些共享数据进行访问。

一个线程在访问共享数据前必须申请相应的锁（许可证），线程的这个动作被称为锁的获得（Acquire）。一个线程获得某个锁（持有许可证），我们就称该线程为相应锁的持有线程（线程持有许可证），一个锁一次只能被一个线程持有。锁的持有线程可以对该锁所保护的共享数据进行访问，访问结束后该线程必须释放（Release）相应的锁。锁的持有线程在其获得锁之后和释放锁之前这段时间内所执行的代码被称为临界区（Critical Section）。因此，共享数据只允许在临界区内进行访问，临界区一次只能被一个线程执行。

约定

如果有多个线程访问同一个锁所保护的共享数据，那么我们就称这些线程同步在这个锁上，或者称我们对这些线程所进行的共享数据访问进行加锁；相应地，这些线程所执行的临界区就被称为这个锁所引导的临界区。

锁具有排他性（Exclusive），即一个锁一次只能被一个线程持有。因此，这种锁被称为排他锁或者互斥锁（Mutex）。这种锁的实现方式代表了锁的基本原理，如图 3-1 所示。本书后续还会提到另外一种锁——读写锁，它可以被看作排他锁的一种相对改进。

按照 Java 虚拟机对锁的实现方式划分，Java 平台中的锁包括内部锁（Intrinsic Lock）和显式锁（Explicit Lock）。内部锁是通过 `synchronized` 关键字实现的；显式锁是通过 `java.concurrent.locks.Lock` 接口的实现类（如 `java.concurrent.locks.ReentrantLock` 类）实现的。

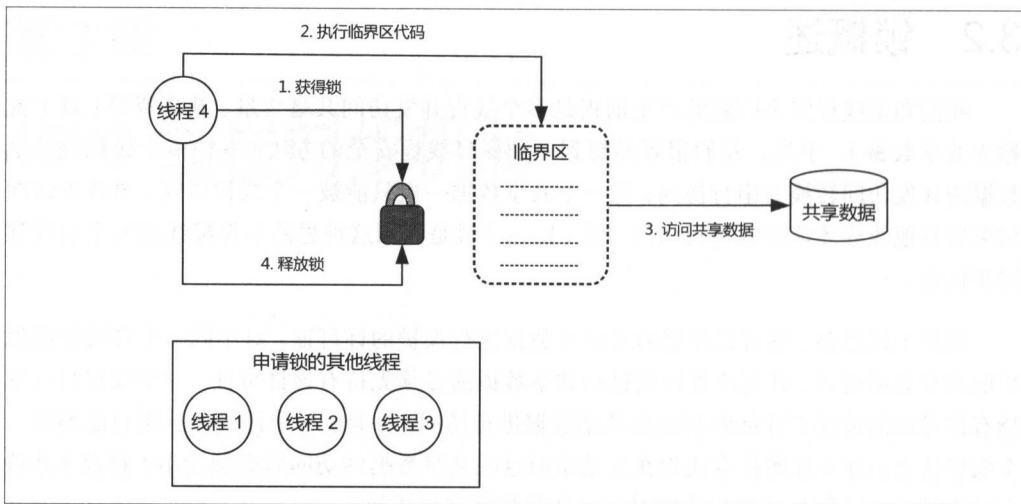


图 3-1 互斥锁示意图

3.2.1 锁的作用

提示

读者也可以先不求甚解地阅读一下本节内容，在学习了后续两节内容之后再回过头来看本节会更容易理解。

锁能够保护共享数据以实现线程安全，其作用包括保障原子性、保障可见性和保障有序性。

锁是通过互斥保障原子性的。所谓互斥（Mutual Exclusion），就是指一个锁一次只能被一个线程持有。因此一个线程持有一个锁的时候，其他线程无法获得该锁，而只能等待其释放该锁后再申请。这就保证了临界区代码一次只能被一个线程执行。因此，一个线程执行临界区期间没有其他线程能够访问相应的共享数据，这使得临界区代码所执行的操作自然而然地具有不可分割的特性，即具备了原子性。

从互斥的角度来看，锁其实是将多个线程对共享数据的访问由本来的并发（未使用锁的情况下）改为串行（使用锁之后）。因此，虽然实现并发是多线程编程的目标，但是这种并发往往是并发中带有串行的局部并发。这好比公路维修使得多股车道在某处被合并成一股小车道，从而使原本在多股车道上并驾齐驱的车辆不得不“鱼贯而行”。

我们知道，可见性的保障是通过写线程冲刷处理器缓存和读线程刷新处理器缓存这两个动作实现的。在 Java 平台中，锁的获得隐含着刷新处理器缓存这个动作，这使得读线

程在执行临界区代码前（获得锁之后）可以将写线程对共享变量所做的更新同步到该线程执行处理器的高速缓存中；而锁的释放隐含着冲刷处理器缓存这个动作，这使得写线程对共享变量所做的更新能够被“推送”到该线程执行处理器的高速缓存中，从而对读线程可同步。因此，锁能够保障可见性。

锁的互斥性及其对可见性的保障合在一起，可保证临界区内的代码能够读取到共享数据的最新值。由于锁的互斥性，同一个锁所保护的共享数据一次只能够被一个线程访问，因此线程在临界区中所读取到共享数据的相对新值（锁对保障可见性的结果）同时也是最新值。

锁不仅能够保障临界区中的代码能够读取到共享变量的最新值¹。对引用型共享变量，锁还可保障临界区中的代码能够读取到该变量所引用对象的字段（实例变量和静态变量）的最新值。这点可以推广到数组变量，即如果共享变量是个数组，那么锁能够保障临界区中的代码可以读取到该数组中各个元素的最新值²。

锁能够保障有序性。写线程在临界区中所执行的一系列操作在读线程所执行的临界区看起来像是完全按照源代码顺序执行的，即读线程对这些操作的感知顺序与源代码顺序一致。这是锁对原子性和可见性的保障的结果。设写线程在临界区中更新了 `b`、`c` 和 `flag` 这 3 个共享变量，如下代码片段所示：

```
b = a + 1;  
c = 2;  
flag = true;
```

由于锁对可见性的保障，写线程在临界区中对上述任何一个共享变量所做的更新都对读线程可见。并且，由于临界区内的操作具有原子性，因此写线程对上述共享变量的更新会同时对读线程可见，即在读线程看来这些变量就像是在同一刻被更新的。因此读线程并无法（也没有必要）区分写线程实际上是以什么顺序更新上述变量的，这意味着读线程可以认为写线程是依照源代码顺序更新上述共享变量的，即有序性得以保障。

由于锁能够保障有序性，因此对于上述例子，可有：如果一个读线程在临界区中读取到变量 `c` 的值为 2，那么 `flag` 的值必然为 `true`，`b` 的值必然比 `a` 的值大 1；如果一个读线程在临界区中读取到变量 `flag` 的值为 `true`，那么 `c` 的值必然为 2，`b` 的值必然比 `a` 的值大 1……

尽管锁能够保障有序性，但是这并不意味着临界区内的内存操作不能够被重排序。临

1 引用型变量的值相当于内存地址，从这点来看引用类型变量与基本类型变量并没有区别。

2 这当然也是有前提的：系统中不能够存在对共享变量所引用的对象（数组）进行不加锁的访问。

界区内的任意两个操作依然可以在临界区之内被重排序（即不会重排到临界区之外）。由于临界区内的操作具有的原子性，写线程在临界区内对各个共享数据的更新同时对读线程可见，因此这种重排序并不会对其他线程产生影响。

在理解以及使用锁保证线程安全的时候，需要注意锁对可见性、原子性和有序性的保障是有条件的，我们要同时保证以下两点得以满足。

- 这些线程在访问同一组共享数据的时候必须使用同一个锁。
- 这些线程中的任意一个线程，即使其仅仅是读取这组共享数据而没有对其进行更新的话，也需要在读取时持有相应的锁。

上述任意一个条件未满足都会使原子性、可见性和有序性没有保障。可见，我们说锁能够保护共享数据其实是一种“协议”的结果，这个协议就是任何访问该共享数据的写线程、读线程都要满足上述条件。只要有任何一个线程没有遵守这个“协议”，这个“协议”实际上就被打破了，从而无法保障线程安全。这就好比交通规则（“协议”）要靠人人都遵守才能保障交通安全一样。

注意 锁对可见性、原子性和有序性的保障是有前提的。因此，访问同一组共享数据的多个线程必须同步在同一锁实例上，并且即使是仅仅对共享数据进行读取（而没有更新）的访问也要加锁。

3.2.2 与锁相关的几个概念

1. 可重入性

可重入性（Reentrancy）描述这样一个问题：一个线程在其持有一个锁的时候能否再次（或者多次）申请该锁。如果一个线程持有一个锁的时候还能够继续成功申请该锁，那么我们就称该锁是可重入的（Reentrant），否则我们就称该锁为非可重入的（Non-reentrant）。可重入性问题的由来可以通过如下伪代码理解：

```
void metheadA() {
    acquireLock(lock); // 申请锁 lock

    // 省略其他代码
    methodB();
    releaseLock(lock); // 释放锁 lock
}

void metheadB() {
```

```

acquireLock(lock); // 申请锁 lock

// 省略其他代码
releaseLock(lock); // 释放锁 lock
}

```

方法 `methodA` 使用了锁 `lock`, 该锁引导的临界区代码又调用了另外一个方法 `methodB`, 而方法 `methodB` 也使用了 `lock`。那么, 这就产生了一个问题: `methodA` 的执行线程持有锁 `lock` 的时候调用了 `methodB`, 而 `methodB` 执行的时候又去申请锁 `lock`, 而 `lock` 此时正被当前线程持有 (未被释放)。那么, 此时 `methodB` 究竟能否获得 (申请成功) `lock` 呢? 可重入性就描述了这样一个问题。

扩展阅读 可重入锁是如何实现的?

可重入锁可以被理解为一个对象, 该对象包含一个计数器属性。计数器属性的初始值为 0, 表示相应的锁还没有被任何线程持有。每次线程获得一个可重入锁的时候, 该锁的计数器值会被增加 1。每次一个线程释放锁的时候, 该锁的计数器属性值就会被减 1。一个可重入锁的持有线程初次获得该锁时相应的开销相对大, 这是因为该锁的持有线程必须与其他线程“竞争”以获得锁。可重入锁的持有线程继续获得相应锁所产生的开销要小得多, 这是因为此时 Java 虚拟机只需要将相应锁的计数器属性值增加 1 即可以实现锁的获得。

2. 锁的争用与调度

锁可以被看作多线程程序访问共享数据时所需持有的一种排他性资源。因此, 资源的争用、调度的概念对锁也是适用的。尽管锁的调度基本上是 Java 虚拟机的设计者需要考虑的事情, 但是适当了解一下这个概念对于今后我们分析和定位多线程问题有所帮助。Java 平台中锁的调度策略也包括公平策略和非公平策略, 相应的锁就被称为公平锁和非公平锁。内部锁属于非公平锁, 而显式锁则既支持公平锁又支持非公平锁。

3. 锁的粒度

一个锁实例可以保护一个或者多个共享数据。一个锁实例所保护的共享数据的数量大小就被称为该锁的粒度 (Granularity)。一个锁实例保护的共享数据的数量大, 我们就称该锁的粒度粗, 否则就称该锁的粒度细。锁粒度的粗细是相对的, 就像美与丑之间的相对性: 有丑的事物才能衬托出美的事物。锁的粒度过粗会导致线程在申请锁的时候需要进行不必要的等待。这好比我们去银行柜台办理业务的情形: 假如一个柜台同时能够办理多种业务, 那么就可能出现这样的场景——办理客户资料变更的客户需要等待前面要办理定期存款的客户。而如果一个柜台只办理一种业务, 比如将开户、销户和客户资料变更归为一种业

务放在一个柜台办理,那么办理客户资料变更的客户需要等待的时间就会相对减少。不过,锁的粒度过细会增加锁调度的开销。

3.2.3 锁的开销及其可能导致的问题

锁的开销包括锁的申请和释放所产生的开销,以及锁可能导致的上下文切换的开销。这些开销主要是处理器时间。

锁可能导致上下文切换。我们知道,多个线程争用排他性资源可能导致上下文切换,因此,锁作为一种排他性资源,一旦被争用就可能导致上下文切换,而没有被争用的锁则可能不会导致上下文切换³。

此外,锁的不正确使用也会导致如下一些线程活性故障。

- 锁泄漏 (Lock Leak)。锁泄漏是指一个线程获得某个锁之后,由于程序的错误、缺陷致使该锁一直无法被释放而导致其他线程一直无法获得该锁的现象。因此,锁泄漏会导致同步在该锁上的所有线程都无法进展。锁泄漏的危害性体现在其不易被发现:可重入锁在争用程度比较低的情况下极有可能只有一个线程反复申请该锁,此时即使这个线程持有该锁之后就一直不释放也不妨碍其后续再次获得该锁(这是由可重入锁本身来保证的);然而,随着争用程度的提高,其他线程也加入申请该锁的行列,这时先前的线程一直未释放锁,这会导致这些线程永远无法获得锁。不幸的是,此时发现问题可能为时已晚——系统可能已经上线运行了!因此,锁泄漏更像是“地雷”,一旦埋下则随时可能会被人踩中而爆炸!
- 锁的不正确使用还可能导致死锁、锁死等线程活性故障,第7章我们会详细讲解这些问题。

3.3 内部锁: synchronized 关键字

Java 平台中的任何一个对象都有唯一一个与之关联的锁。这种锁被称为监视器 (Monitor) 或者内部锁 (Intrinsic Lock)。内部锁是一种排他锁,它能够保障原子性、可见性和有序性。

内部锁是通过 `synchronized` 关键字实现的。`synchronized` 关键字可以用来修饰方法以及代码块 (花括号 “{}” 包裹的代码)。

³ 在 Java 中,被争用的锁不一定会导致上下文切换。参见 12.1.4 节。

`synchronized` 关键字修饰的方法就被称为同步方法 (Synchronized Method)。 `synchronized` 修饰的静态方法就被称为同步静态方法, `synchronized` 修饰的实例方法就被称为同步实例方法。同步方法的整个方法体就是一个临界区。

第2章中介绍的循环递增序列号生成器就是通过使用同步方法实现线程安全的,如清单3-1所示。

清单 3-1 同步方法实例

```
public class SafeCircularSeqGenerator implements CircularSeqGenerator {
    private short sequence = -1;
    public synchronized short nextSequence() {
        if (sequence >= 999) {
            sequence = 0;
        } else {
            sequence++;
        }
        return sequence;
    }
}
```

由于 `nextSequence()` 会被其多个客户端线程执行, 因此 `sequence` 实例变量就成为这些线程的共享数据。`synchronized` 会确保 `nextSequence()` 一次只能够被一个线程执行 (锁的排他性), 因此 `nextSequence()` 的客户端线程在执行该方法时实际上是串行的 (整体并发中的局部串行), 这就排除了对 `sequence` 变量访问操作的交错的可能性。另外, 锁对可见性的保障使得 `nextSequence()` 的当前执行线程对 `sequence` 变量的更新对该方法的下一个执行线程可见。由此, 我们保障了线程安全 (保障原子性和可见性), 从而避免了序列号生成时的重号 (读取旧数据) 和丢号 (丢失更新) 的问题。

`synchronized` 关键字修饰的代码块被称为同步块 (Synchronized Block), 其语法如下所示:

```
synchronized(锁句柄) {
    // 在此代码块中访问共享数据
}
```

`synchronized` 关键字所引导的代码块就是临界区。锁句柄是一个对象的引用 (或者能够返回对象的表达式)。例如, 锁句柄可以填写为 `this` 关键字 (表示当前对象)。习惯上我们也直接称锁句柄为锁。锁句柄对应的监视器就被称为相应同步块的引导锁。相应地, 我们称呼相应的同步块为该锁引导的同步块。

同步实例方法相当于以“this”为引导锁的同步块⁴。因此，清单 3-1 中的同步方法可以改写为：

```
public short nextSequence() {
    synchronized (this) {
        if (sequence >= 999) {
            sequence = 0;
        } else {
            sequence++;
        }
        return sequence;
    }
}
```

作为锁句柄的变量通常采用 `final` 修饰。这是因为锁句柄变量的值一旦改变，会导致执行同一个同步块的多个线程实际上使用不同的锁，从而导致竞态。有鉴于此，通常会使用 `private` 修饰作为锁句柄的变量。

注意 作为锁句柄的变量通常采用 `private final` 修饰，如：`private final Object lock = new Object();`

同步静态方法相当于以当前类对象（Java 中的类本身也是一个对象）为引导锁的同步块。例如同步静态方法：

```
public class SynchronizedMethodExample {
    public static synchronized void staticMethod() {
        // 在此访问共享数据
    }
    // ....
}
```

相当于：

```
public class SynchronizedMethodExample {
    public static void staticMethod() {
        synchronized (SynchronizedMethodExample.class) {
            // 在此访问共享数据
        }
    }
    // ....
}
```

⁴ 实际上，Java 虚拟机及编译器对同步块和同步方法的处理方式是不同的，但是这并不影响我们做出这样的理解。

线程在执行临界区代码的时候必须持有该临界区的引导锁。一个线程执行到同步块（同步方法也可看作同步块）时必须先申请该同步块的引导锁，只有申请成功（获得）该锁的线程才能够执行相应的临界区。一个线程执行完临界区代码后引导该临界区的锁就会被自动释放。在这个过程中，线程对内部锁的申请与释放的动作由 Java 虚拟机负责代为实施，这也正是 `synchronized` 实现的锁被称为内部锁的原因。

内部锁的使用并不会导致锁泄漏。这是因为 Java 编译器（`javac`）在将同步块代码编译为字节码的时候，对临界区中可能抛出的而程序代码中又未捕获的异常进行了特殊（代）处理，这使得临界区的代码即使抛出异常也不会妨碍内部锁的释放。

内部锁的调度

Java 虚拟机会为每个内部锁分配一个入口集（`Entry Set`），用于记录等待获得相应内部锁的线程。多个线程申请同一个锁的时候，只有一个申请者能够成为该锁的持有线程（即申请锁的操作成功），而其他申请者的申请操作会失败。这些申请失败的线程并不会抛出异常，而是会被暂停（生命周期状态变为 `BLOCKED`）并被存入相应锁的入口集中等待再次申请锁的机会。入口集中的线程就被称为相应内部锁的等待线程。当这些线程申请的锁被其持有线程释放的时候，该锁的入口集中的一个任意线程会被 Java 虚拟机唤醒，从而得到再次申请锁的机会。由于 Java 虚拟机对内部锁的调度仅支持非公平调度，被唤醒的等待线程占用处理器运行时可能还有其他新的活跃线程（处于 `RUNNABLE` 状态，且未进入过入口集）与该线程抢占这个被释放锁，因此被唤醒的线程不一定就能成为该锁的持有线程。另外，Java 虚拟机如何从一个锁的入口集中选择一个等待线程，作为下一个可以参与再次申请相应锁的线程，这个细节与 Java 虚拟机的具体实现有关：这个被选中的线程有可能是入口集中等待时间最长的线程，也可能是等待时间最短的线程，或者完全是随机的一个线程。因此，我们不能依赖这个具体的选择算法。

3.4 显式锁：Lock 接口

显式锁是自 JDK 1.5 开始引入的排他锁。作为一种线程同步机制，其作用与内部锁相同。它提供了一些内部锁所不具备的特性，但并不是内部锁的替代品。

显式锁（`Explicit Lock`）是 `java.util.concurrent.locks.Lock` 接口的实例。该接口对显式锁进行了抽象，其定义的方法如图 3-2 所示。类 `java.util.concurrent.locks.ReentrantLock` 是 `Lock` 接口的默认实现类。

方法摘要	
void	lock() 获取锁。
void	lockInterruptibly() 如果当前线程未被中断，则获取锁。
Condition	newCondition() 返回绑定到此 Lock 实例的新 Condition 实例。
boolean	tryLock() 仅在调用时锁为空闲状态才获取该锁。
boolean	tryLock(long time, TimeUnit unit) 如果锁在给定的等待时间内空闲，并且当前线程未被中断，则获取锁。
void	unlock() 释放锁。

图 3-2 Lock 接口定义的方法

一个 Lock 接口实例就是一个显式锁对象，Lock 接口定义的 lock 方法和 unlock 方法分别用于申请和释放相应 Lock 实例表示的锁。显式锁的使用方法如下所示：

```
private final Lock lock=...; // 创建一个 Lock 接口实例
.....

lock.lock(); // 申请锁 lock
try{
    // 在此对共享数据进行访问
    .....
}finally{
    // 总是在 finally 块中释放锁，以避免锁泄漏
    lock.unlock(); // 释放锁 lock
}
```

显式锁的使用包括以下几个方面。

- 创建 Lock 接口的实例。如果没有特别的要求，我们就可以创建 Lock 接口的默认实现类 ReentrantLock 的实例作为显式锁使用。从字面上可以看出 ReentrantLock 是一个可重入锁。
- 在访问共享数据前申请相应的显式锁。这一步，我们直接调用相应 Lock.lock()即可。
- 在临界区中访问共享数据。Lock.lock()调用与 Lock.unlock()调用之间的代码区域为临界区。不过，一般我们视上述的 try 代码块为临界区。因此，对共享数据的访问都仅放在该代码块中。
- 共享数据访问结束后释放锁。虽然释放锁的操作通过调用 Lock.unlock()即可实现，但是为了避免锁泄漏，我们必须将这个调用放在 finally 块中执行。这样，无论是

临界区代码执行正常结束还是由于其抛出异常而提前退出，相应锁的 `unlock` 方法总是可以被执行，从而避免了锁泄漏。可见，显式锁不像内部锁那样可以由编译器代为规避锁泄漏问题。

我们可以将清单 3-1 中的循环递增序列号生成器使用显式锁来改写，如清单 3-2 所示。

清单 3-2 使用显式锁实现循环递增序列号生成器

```
public class LockbasedCircularSeqGenerator implements CircularSeqGenerator {
    private short sequence = -1;
    private final Lock lock = new ReentrantLock();

    @Override
    public short nextSequence() {
        lock.lock();
        try {
            if (sequence >= 999) {
                sequence = 0;
            } else {
                sequence++;
            }
            return sequence;
        } finally {
            lock.unlock();
        }
    }
}
```

3.4.1 显式锁的调度

`ReentrantLock` 既支持非公平锁也支持公平锁。`ReentrantLock` 的一个构造器的签名如下：

```
ReentrantLock(boolean fair)
```

该构造器使得我们在创建显式锁实例的时候可以指定相应的锁是否是公平锁（`fair` 参数值 `true` 表示是公平锁）。

公平锁保障锁调度的公平性往往是以增加了线程的暂停和唤醒的可能性，即增加了上下文切换为代价的。因此，公平锁适合于锁被持有的时间相对长或者线程申请锁的平均间隔时间相对长的情形。总的来说使用公平锁的开销比使用非公平锁的开销要大，因此显式锁默认使用的是非公平调度策略。

3.4.2 显式锁与内部锁的比较

显式锁和内部锁二者的关系是“尺有所短，寸有所长”——其各自适用场景不同，而不是相互替代。

内部锁是基于代码块的锁，因此其使用基本无灵活性可言：要么使用它，要么不使用它，除此之外别无他选。而显式锁是基于对象的锁，其使用可以充分发挥面向对象编程的灵活性。而内部锁从代码角度看仅仅是一个关键字，它无法充分发挥面向对象编程的灵活性。比如，内部锁的申请与释放只能是在一个方法内进行（因为代码块无法跨方法），而显式锁支持在一个方法内申请锁，却在另外一个方法里释放锁。

内部锁基于代码块的这个特征也使其具有一个优势：简单易用，且不会导致锁泄漏。另外，遗留系统（Legacy System）往往存在大量内部锁的使用（因为这些系统开发的时候还没有显式锁）。而显式锁容易被错用而导致锁泄漏，因此使用显式锁的时候必须注意将锁的释放操作放在 `finally` 块中。然而，这一点却很容易被新手甚至一些“老手”忽略。

显式锁支持了一些内部锁所不支持的特性，这里不一一列举，仅介绍其中的一部分，另外一些特性在后续章节中会体现出来。

如果一个内部锁的持有线程一直不释放这个锁（这通常是由于代码错误导致的），那么同步在该锁之上的所有线程就会一直被暂停而使其任务无法进展。而显式锁则可以轻松地避免这样的问题。`Lock` 接口定义了一个 `tryLock` 方法。该方法的作用是尝试申请相应 `Lock` 实例锁表示的锁。如果相应的锁未被其他任何线程持有，那么该方法会返回 `true`，表示其获得了相应的锁；否则，该方法并不会导致其执行线程被暂停而是直接返回 `false`，表示其未获得相应的锁。`tryLock` 方法的使用方法如下代码模板所示：

```
Lock lock = ...;
if (lock.tryLock()) {
    try {
        // 在此访问共享数据
    } finally {
        lock.unlock();
    }
} else {
    // 执行其他操作
}
```

`tryLock` 方法是个多载（Overload）的方法，它还有另外一个签名版本：

```
boolean tryLock(long time, TimeUnit unit)
```

这个版本的 `tryLock` 方法使得我们可以指定一个时间。如果当前线程没有在指定的时间内成功申请到（获得）相应的锁，那么 `tryLock` 方法就直接返回 `false`。

在锁的调度方面，内部锁仅支持非公平锁；而显式锁既支持非公平锁，又支持公平锁。

在问题定位方面，尤其是定位生产环境上的问题的时候，线程转储（Thread dump，参见第1章）就像是线程的“工作报告”一样可以告诉我们 Java 虚拟机中关于线程的详细信息。线程转储中会包含内部锁的相关信息，包括一个线程等待哪些锁以及这些锁的当前（获取相应线程转储那一刻）持有线程。而在 JDK 1.5 下，线程转储中并不包含显式锁的相关信息。不过，JDK 1.6 提供的工具 `jstack` 所产生的线程转储中可以包含显式锁的信息⁵。例如，运行如清单 3-3 所示的程序，并获取该程序的线程转储。

清单 3-3 演示线程转储显式锁信息的示例程序

```
public class ExplicitLockInfo {
    private static final Lock lock = new ReentrantLock();
    private static int sharedData = 0;

    public static void main(String[] args) throws Exception {
        Thread t = new Thread(new Runnable() {
            @Override
            public void run() {
                lock.lock();
                try {
                    try {
                        Thread.sleep(2200000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    sharedData = 1;
                } finally {
                    lock.unlock();
                }
            }
        });
        t.start();
        Thread.sleep(100);
        lock.lock();
        try {
```

5 获取线程转储的时候需要，指定 `jstack` 的“-l”参数才能使产生的线程转储包含显式锁的相关信息。在 Linux 系统中使用“kill -3 Java 进程 PID”命令所产生的线程转储并不会显示某个显式锁实例是被哪个线程持有的。

```
        System.out.println("sharedData:" + sharedData);
    } finally {
        lock.unlock();
    }
}
}
```

可以得到类似如下的线程信息（省略部分输出）⁶：

Full thread dump Java HotSpot(TM) 64-Bit Server VM (20.45-b01 mixed mode):

"Thread-1" prio=10 tid=0x00007fe3e81e8800 nid=0x2bb1 waiting on condition [0x00007fe3e3ce29000]

```
java.lang.Thread.State: TIMED_WAITING (sleeping)
    at java.lang.Thread.sleep(Native Method)
    at io.github.viscent.mtia.ch3.ExplicitLockInfo$1.run(ExplicitLockInfo.java:19)
    at java.lang.Thread.run(Thread.java:662)
```

Locked ownable synchronizers:

- <0x00000000d7291ea8> (aj.u.c.locks.ReentrantLock\$NonfairSync)

……省略其他输出

"main" prio=10 tid=0x00007fe3e8008000 nid=0x2b9d waiting on condition [0x00007fe3efcc4000]

```
java.lang.Thread.State: WAITING (parking)
    at sun.misc.Unsafe.park(Native Method)
    - parking to wait for <0x00000000d7291ea8> (a j.u.c.locks.ReentrantLock$NonfairSync)
    at j.u.c.locks.LockSupport.park(LockSupport.java:156)
    at j.u.c.locks.AbstractQueuedSynchronizer.parkAndCheckInterrupt(AbstractQueuedSynchronizer.java:811)
    at j.u.c.locks.AbstractQueuedSynchronizer.acquireQueued(AbstractQueuedSynchronizer.java:842)
    at j.u.c.locks.AbstractQueuedSynchronizer.acquire(AbstractQueuedSynchronizer.java:1178)
    at j.u.c.locks.ReentrantLock$NonfairSync.lock(ReentrantLock.java:186)
    at j.u.c.locks.ReentrantLock.lock(ReentrantLock.java:262)
    at io.github.viscent.mtia.ch3.ExplicitLockInfo.main(ExplicitLockInfo.java:38)
```

Locked ownable synchronizers:

- None

……省略其他输出

可见，main 线程由于需要获得唯一标识为“0x00000000d7291ea8”的显式锁而处于等待状态，而这个锁被线程“Thread-1”所持有。

6 为了便于排版，笔者把线程转储中真实的包名 java.util.concurrent.locks 替换成 j.u.c.locks。

显式锁提供了一些接口（指方法）可以用来对锁的相关信息进行监控，而内部锁不支持这种特性。ReentrantLock 中定义的方法 isLocked() 可用于检测相应锁是否被某个线程持有⁷，getQueueLength() 方法可用于检查相应锁的等待线程的数量。

显式锁与内部锁在性能方面的差异主要包括：

- Java 1.6/1.7 对内部锁做了一些优化，这些优化在特定情况下可以减少锁的开销。这些优化包括锁消除（Lock Elimination）、锁粗化（Lock Coarsening）、偏向锁（Biased Lock）和适配性锁（Adaptive Lock），详情可参见第 12 章。而这些优化在 Java 1.6/1.7 中并没有运用到显式锁上。不过，这并不排除 Java 的后续版本会在显式锁中引入这些优化（可能只是部分）⁸。
- 在 Java 1.5 中，在高争用的情况下，内部锁的性能急剧下降，而显式锁的性能下降则少得多。换言之，Java 1.5 中显式锁的可伸缩性（Scalability）比内部锁的可伸缩性要好。到了 Java 1.6，随着 JDK 对内部锁所做的一些改进，显式锁和内部锁之间的可伸缩性差异已经变得非常小了。

3.4.3 内部锁还是显式锁：锁的选用

内部锁的优点是简单易用，显式锁的优点是功能强大，这两种锁各自都存在一些弱势。

一般来说，新开发的代码中我们可以选用显式锁。但是选用显式锁的时候需要注意：显式锁的不正确使用会导致锁泄漏这样严重的问题；线程转储可能无法包含显式锁的相关信息，从而导致问题定位的困难。

另外，我们也可以使用相对保守的策略——默认情况下选用内部锁，仅在需要显式锁所提供的特性的时候才选用显式锁。比如，在多数线程持有一个锁的时间相对长或者线程申请锁的平均时间间隔相对长的情况下使用非公平锁是不太恰当的，因此我们可考虑使用公平锁（显式锁）。

*3.4.4 改进型锁：读写锁

锁的排他性使得多个线程无法以线程安全的方式在同一时刻对共享变量进行读取（只是读取而不更新），这不利于提高系统的并发性。

7 Thread.holdsLock(Object) 只能检测当前线程是否持有指定的内部锁。

8 出自 Oracle Labs 资深研究科学家（Senior Research Scientist）David Dice 的观点，见：https://blogs.oracle.com/dave/entry/java_util_concurrent_reentrantlock_vs。

术语
定义

对于同步在同一锁之上的线程而言，对共享变量仅进行读取而没有进行更新的线程被称为只读线程，简称读线程。对共享变量进行更新（包括先读取后更新）的线程就被称为写线程。

读写锁（Read/Write Lock）是一种改进型的排他锁，也被称为共享/排他（Shared/Exclusive）锁。读写锁允许多个线程可以同时读取（只读）共享变量，但是一次只允许一个线程对共享变量进行更新（包括读取后再更新）。任何线程读取共享变量的时候，其他线程无法更新这些变量；一个线程更新共享变量的时候，其他任何线程都无法访问该变量。

读写锁的功能是通过其扮演的两种角色——读锁（Read Lock）和写锁（Write Lock）实现的，如表 3-1 所示。读线程在访问共享变量的时候必须持有相应读写锁的读锁。读锁是可以同时被多个线程持有的，即读锁是共享的（Shared），一个读线程持有一个读锁的时候并不妨碍其他读线程获得该读锁。写线程在访问共享变量的时候必须持有相应读写锁的写锁。写锁是排他的（Exclusive），即一个线程持有写锁的时候其他线程无法获得相应锁的写锁或读锁。因此，写锁保障了写线程对共享变量的访问（包括更新）是独占的。读锁实际上只是在读线程之间是共享的，任何一个线程持有一个读锁的时候，其他任何线程都无法获得相应锁的写锁。这就保障了读线程在读取共享变量期间没有其他线程能够对这些变量进行更新，从而使读线程能够读取到相应变量的最新值。总的来说，读锁对于读线程来说起到保护其访问的共享变量在其访问期间不被修改的作用，并使多个读线程可以同时读取这些变量从而提高了并发性；而写锁保障了写线程能够以独占的方式安全地更新共享变量。写线程对共享变量的更新对读线程是可见的。

表 3-1 读写锁的两种角色

	获得条件	排他性	作用
读锁	相应的写锁未被任何线程持有	对读线程是共享的，对写线程是排他的	允许多个读线程可以同时读取共享变量，并保障读线程读取共享变量期间没有其他任何线程能够更新这些共享变量
写锁	该写锁未被其他任何线程持有并且相应的读锁未被其他任何线程持有	对写线程和读线程都是排他的	使得写线程能够以独占的方式访问共享变量

java.util.concurrent.locks.ReadWriteLock 接口是对读写锁的抽象，其默认实现类是 java.util.concurrent.locks.ReentrantReadWriteLock。ReadWriteLock 接口定义了两个方法：readLock()和 writeLock()，如图 3-3 所示，分别用于返回相应读写锁实例的读锁和写锁。这两个方法的返回值类型都是 Lock，这并不是表示一个 ReadWriteLock 接口实例对应两个

锁，而是代表一个 `ReadWriteLock` 接口实例可以充当两种角色。这就好比一个人在不同场合（场所）中所扮演的两种不同角色——小明的爸爸（人）是个医生，他在家的时候更多的是扮演父亲的角色，在医院的时候更多的是扮演医务工作者的角色。显然，我们不能据此断定小明的爸爸是两个人。

方法摘要	
Lock	<code>readLock()</code> 返回用于读取操作的锁。
Lock	<code>writeLock()</code> 返回用于写入操作的锁。

图 3-3 `ReadWriteLock` 接口定义的方法

读写锁的使用方法与显式锁相似，也要注意锁泄漏问题，如清单 3-4 所示。

清单 3-4 读写锁使用方法

```
public class ReadWriteLockUsage {
    private final ReadWriteLock rwLock = new ReentrantReadWriteLock();
    private final Lock readLock = rwLock.readLock();
    private final Lock writeLock = rwLock.writeLock();

    // 读线程执行该方法
    public void reader() {
        readLock.lock(); // 申请读锁
        try {
            // 在此区域读取共享变量
        } finally {
            readLock.unlock(); // 总是在 finally 块中释放锁，以免锁泄漏
        }
    }

    // 写线程执行该方法
    public void writer() {
        writeLock.lock(); // 申请读锁
        try {
            // 在此区域访问（读、写）共享变量
        } finally {
            writeLock.unlock(); // 总是在 finally 块中释放锁，以免锁泄漏
        }
    }
}
```

与普通的排他锁（如内部锁和 `ReentrantLock`）相比，读写锁在排他性方面比较弱（这是我们所期望的）。在原子性、可见性和有序性保障方面，它所起到的作用与普通的排他锁是一致的。写线程释放写锁所起到的作用相当于一个线程释放一个普通排他锁；读线程

获得读锁所起到的作用相当于一个线程获得一个普通排他锁。由于读写锁内部实现比内部锁和其他显式锁要复杂得多，因此读写锁适合于在以下条件同时得以满足的场景中使用：

- 只读操作比写（更新）操作要频繁得多；
- 读线程持有锁的时间比较长。

只有同时满足上面两个条件的时候，读写锁才是适宜的选择。否则，使用读写锁会得不偿失（开销）。

`ReentrantReadWriteLock` 所实现的读写锁是个可重入锁。`ReentrantReadWriteLock` 支持锁的降级（Downgrade），即一个线程持有读写锁的写锁的情况下可以继续获得相应的读锁，如清单 3-5 所示。

清单 3-5 读写锁的降级示例

```
public class ReadWriteLockDowngrade {
    private final ReadWriteLock rwLock = new ReentrantReadWriteLock();
    private final Lock readLock = rwLock.readLock();
    private final Lock writeLock = rwLock.writeLock();

    public void operationWithLockDowngrade() {
        boolean readLockAcquired = false;
        writeLock.lock(); // 申请写锁
        try {
            // 对共享数据进行更新
            // ...
            // 当前线程在持有写锁的情况下申请读锁 readLock
            readLock.lock();
            readLockAcquired = true;
        } finally {
            writeLock.unlock(); // 释放写锁
        }

        if (readLockAcquired) {
            try {
                // 读取共享数据并据此执行其他操作
                // ...

            } finally {
                readLock.unlock(); // 释放读锁
            }
        } else {
            // ...
        }
    }
}
```

锁的降级的反面是锁的升级 (Upgrade)，即一个线程在持有读写锁的读锁的情况下，申请相应的写锁。ReentrantReadWriteLock 并不支持锁的升级。读线程如果要转而申请写锁，需要先释放读锁，然后申请相应的写锁。

3.5 锁的适用场景

锁是 Java 线程同步机制中功能最强大、适用范围最广泛，同时也是开销最大、可能导致的问题最多的同步机制。多个线程共享同一组数据的时候，如果其中有线程涉及如下操作，那么我们就可以考虑使用锁。

- check-then-act 操作：一个线程读取共享数据并在此基础上决定其下一个操作是什么。
- read-modify-write 操作：一个线程读取共享数据并在此基础上更新该数据。不过，某些像自增操作 (“count++”) 这种简单的 read-modify-write 操作，我们可以使用本章后续内容介绍的原子变量类来实现线程安全。
- 多个线程对多个共享数据进行更新：如果这些共享数据之间存在关联关系，那么为了保障操作的原子性我们可以考虑使用锁。例如，关于服务器的配置信息可能包括主机 IP 地址、端口号等。一个线程如果要对这些数据进行更新，则必须要保障更新操作的原子性，即主机 IP 地址和端口号总是一起被更新的，否则其他线程可能看到一个并不真实存在的主机 IP 地址和端口号组合所代表的服务器。本章的后续内容也会介绍一种该场景下的替代线程同步机制。

3.6 线程同步机制的底层助手：内存屏障

约定

对于同步在同一个锁之上的多个线程，我们称对共享变量进行更新的线程为写线程，对共享变量进行读取的线程为读线程。因此，一个线程可以既是写线程又是读线程。读线程、写线程在访问共享变量时必须持有相应的锁。

前文（参见 3.2.1 节）我们讲解锁是如何保证可见性的时候提到了线程获得和释放锁时所分别执行的两个动作：刷新处理器缓存和冲刷处理器缓存。对于同一个锁所保护的共享数据而言，前一个动作保证了该锁的当前持有线程能够读取到前一个持有线程对这些数据所做的更新，后一个动作保证了该锁的持有线程对这些数据所做的更新对该锁的后续持有线程可见。那么，这两个动作是如何实现的呢？弄清楚这个问题有助于我们学习和掌握包括锁在内的所有 Java 线程同步机制。

Java 虚拟机底层实际上是借助内存屏障（Memory Barrier，也称 Fence）来实现上述两个动作的⁹。内存屏障是对一类仅针对内存读、写操作指令（Instruction）的跨处理器架构（比如 x86、ARM）的比较底层的抽象（或者称呼）。内存屏障是被插入到两个指令之间进行使用的，其作用是禁止编译器、处理器重排序从而保障有序性。它在指令序列（如指令 1；指令 2；指令 3）中就像是一堵墙（因此被称为屏障）一样使其两侧（之前和之后）的指令无法“穿越”它（一旦穿越了就是重排序了）。但是，为了实现禁止重排序的功能，这些指令也往往具有一个副作用——刷新处理器缓存、冲刷处理器缓存，从而保证可见性。不同微架构的处理器所提供的这样的指令是不同的，并且出于不同的目的使用的相应指令也是不同的。例如对于“写—写”（写后写）操作，如果仅仅是为了防止（禁止）重排序而对可见性保障没有要求，那么在 x86 架构的处理器下使用空操作就可以了（因为 x86 处理器不会对“写—写”操作进行重排序）¹⁰。而如果对可见性有要求（比如前一个写操作的结果要在后一个写操作执行前对其他处理器可见），那么在 x86 处理器下需要使用 LOCK 前缀指令或者 sfence 指令、mfence 指令；在 ARM 处理器下则需要使用 DMB 指令。

约定

本书所说的读操作和写操作没有特别说明的，都是指对主内存（即 DRAM）进行的读、写操作。具体地，读操作（Load 或者 Read）指将主内存中的数据（通过高速缓存）读取到寄存器中，如下 x86 汇编代码所示：

```
mov    edx, 0f80f802ch ; 将内存地址 0f80f802ch 中的内容读取到寄存器 edx 中
```

写操作（Store 或者 Write）指将数据写到共享内存中，如下 x86 汇编代码所示：

```
mov    0f80f802ch,edx ; 将寄存器 edx 中的内容写入地址 0f80f802ch 所指示的内存
```

指令序列：若干指令的集合。例如，上面的一个读操作和写操作组合在一起就形成一个指令序列。

由于内部锁的申请与释放对应的 Java 虚拟机字节码指令分别是 monitorenter 和 monitorexit，因此习惯上我们用 MonitorEnter 表示锁的申请，用 MonitorExit 表示锁的释放。

按照内存屏障所起的作用来划分，本书将内存屏障划分为以下几种。

- 按照可见性保障来划分，内存屏障可分为加载屏障（Load Barrier）和存储屏障（Store Barrier）。加载屏障的作用是刷新处理器缓存，存储屏障的作用冲刷处理

9 许多资料对内存屏障这个概念的描述都是不全面的，甚至是相互矛盾的。这里我们介绍这个概念的目标是希望能够帮助读者深入理解包括锁在内的所有 Java 线程同步机制。读者不必太拘泥于称呼，而是要将关注点放在内存屏障的功能是什么以及这些功能和 Java 的线程同步机制之间的关系方面。

10 这里是指 Java 虚拟机自身需要直接使用内存屏障，而不是指我们编写 Java 应用代码的时候需要直接使用内存屏障。

器缓存。Java 虚拟机会在 `MonitorExit`（释放锁）对应的机器码指令之后插入一个存储屏障，这就保障了写线程在释放锁之前在临界区中对共享变量所做的更新对读线程的执行处理器来说是可同步的；相应地，Java 虚拟机会在 `MonitorEnter`（申请锁）对应的机器码指令之后临界区开始之前的地方插入一个加载屏障，这使得读线程的执行处理器能够将写线程对相应共享变量所做的更新从其他处理器同步到该处理器的高速缓存中。因此，可见性的保障是通过写线程和读线程成对地使用存储屏障和加载屏障实现的。这有点像高考成绩查询：高考阅卷完毕之后虽然每个考生的分数都已经确定，但是这些分数对于考生来说仍然是未知的。只有当分数公布的时候，考生才能够去查询自己的分数。这里，考试成绩就相当于阅卷方和考生之间需要共享的数据：阅卷方公布成绩相当于执行存储屏障，它使得考生的分数可以被查询；考生查询其考试成绩相当于执行加载屏障，它使得考生可以得知自己的考试成绩。

- 按照有序性保障来划分，内存屏障可以分为获取屏障（`Acquire Barrier`）和释放屏障（`Release Barrier`）。获取屏障的使用方式是在一个读操作（包括 `Read-Modify-Write` 以及普通的读操作）之后插入该内存屏障，其作用是禁止该读操作与其后的任何读写操作之间进行重排序，这相当于在进行后续操作之前先要获得相应共享数据的所有权（这也是该屏障的名称来源）。释放屏障的使用方式是在一个写操作之前插入该内存屏障，其作用是禁止该写操作与其前面的任何读写操作之间进行重排序。这相当于在对相应共享数据操作结束后释放所有权（这也是该屏障的名称来源）。Java 虚拟机会在 `MonitorEnter`（它包含了读操作）对应的机器码指令之后临界区开始之前的地方插入一个获取屏障，并在临界区结束之后 `MonitorExit`（它包含了写操作）对应的机器码指令之前的地方插入一个释放屏障。因此，这两种屏障就像是三明治的两层面包片把火腿夹住一样把临界区中的代码（指令序列）包括起来，如图 3-4 所示。

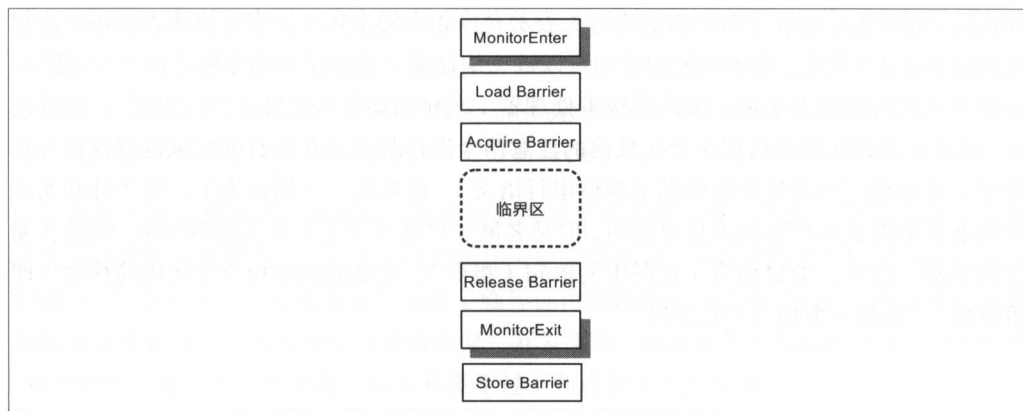


图 3-4 内存屏障在锁中的使用

由于获取屏障禁止了临界区中的任何读、写操作被重排序到临界区之前的可能性，而释放屏障又禁止了临界区中的任何读、写操作被重排序到临界区之后的可能性，因此临界区内的任何读、写操作都无法被重排序到临界区之外。在锁的排他性的作用下，这使得临界区中执行的操作序列具有原子性。因此，写线程在临界区中对各个共享变量所做的更新会同时对读线程可见，即在读线程看来各个共享变量就像是“一下子”被更新的，于是这些线程无从（也无必要）区分这些共享变量是以何种顺序被更新的。这使得写线程在临界区中执行的操作自然而然地具有有序性——读线程对这些操作的感知顺序与源代码顺序一致。

可见，锁对有序性的保障是通过写线程和读线程配对使用释放屏障与加载屏障实现的。

为了保障线程安全，我们需要使用 Java 线程同步机制，而内存屏障则是 Java 虚拟机在实现 Java 线程同步机制时所使用的具体“工具”。因此，Java 应用开发人员一般无须（也不能）直接使用内存屏障。不过，JSR 166 所定义的 Java Fences API（`java.util.concurrent.atomic.Fences`）使得在 Java 语言这一层使用内存屏障成为可能¹¹。不过，本书写作之时 Fences API 尚未发布到任何一个版本的 JDK 之中。

*3.7 锁与重排序

为了使锁能够起到其预定的作用并且尽量避免对性能造成“伤害”，编译器（基本上指 JIT 编译器）和处理器必须遵守一些重排序规则，这些重排序规则禁止一部分的重排并且允许另外一部分的重排序（以便不“伤害”性能）。

总的来说，与锁有关的重排序规则可以理解为语句（指令）相对于临界区的“许进不许出”，如图 3-5 所示（图中的实线箭头表示允许相应的重排序，虚线箭头表示不允许相应的重排序）。可见，临界区外的语句可以被（编译器）重排序到临界区之内（“许进”），而临界区内的操作无法被（编译器或者处理器）重排到临界区之外（“不许出”）；临界区内、临界区前和临界区后这 3 个区域内的任意两个操作都可以在各自的区域范围内进行重排序（只要相应的重排序能够满足貌似串行语义）。这好比一个捕鼠笼子，笼子外和笼子内的老鼠可以在各自区域内自由活动，但是老鼠一旦进入了笼子则无法再出来，只能在笼子内活动。这里，老鼠相当于代码中的语句（指令）、老鼠的活动相当于代码的移动（即重排序），而笼子则相当于临界区。

11 参见：<http://download.java.net/lambda/b95/docs/api/java/util/concurrent/atomic/Fences.html>。

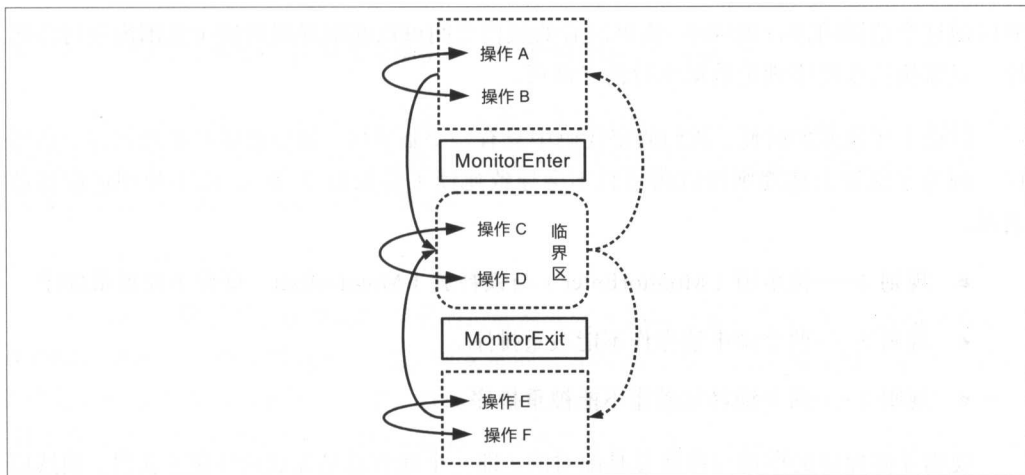


图 3-5 锁与重排序示意图

具体来说，无论是编译器还是处理器，均还需要遵守以下重排序规则。

- 规则 1——临界区内的操作不允许被重排序到临界区之外（即临界区前或者临界区后）。
- 规则 2——临界区内的操作之间允许被重排序。
- 规则 3——临界区外（临界区前或者临界区后）的操作之间可以被重排序。

规则 1 比较容易理解。该规则是锁保障原子性和可见性的基础，编译器和处理器都必须遵守该规则。Java 虚拟机会在临界区的开始之前和结束之后分别插入一个获取屏障和释放屏障，从而禁止临界区内的操作被排到临界区之前和之后。我们也可以从程序语义的角度去理解这个规则：首先，把读操作放在临界区内，说明我们希望该操作能够读取到相应变量的最新值。而如果将这种操作重排到临界区之前或者之后，则可能导致竞态，而使相应的操作可能读取到脏数据。因此，临界区内的读操作不允许被重排到临界区之外。类似地，读者也可从程序语义的角度出发自行分析写操作不允许被重排序到临界区之外。

规则 2 也不难理解。该规则一定程度上避免了对性能造成“伤害”，毕竟重排序有利于发挥处理器的计算性能。临界区内的操作可以被重排序，只要这些重排序是满足貌似串行语义的。这是因为，锁的排他性保证了临界区内的操作是一个原子操作，因此这些操作从其他同步在这个锁的线程的角度看像是一起发生的，故而也就无所谓顺序了：一个操作（原子操作）是无所谓顺序的，它只有发生过了和尚未发生的区别。

规则 3 也是为了避免对性能造成“伤害”。该规则表明临界区前的操作之间可以在临

界区前这个范围内进行重排序，临界区后的操作之间可以在临界区后这个范围内进行重排序，只要执行重排序满足貌似串行语义即可。

讨论上述规则的时候，我们假定代码中只有一个临界区。如果把情形扩展到多个临界区，则为了保证上述规则得以满足且不会导致死锁（参见第 7 章），以下规则还应该被满足。

- 规则 4——锁申请（MonitorEnter）与锁释放（MonitorExit）操作不能被重排序。
- 规则 5——两个锁申请操作不能被重排序。
- 规则 6——两个锁释放操作不能被重排序。

规则 4 确保锁的申请与释放总是配对的，即一个线程总是先成功申请（获得）锁然后才能释放锁，这样才能确保锁这一概念的正确实现。

规则 4、规则 5 和规则 6 合在一起确保了 Java 语义支持嵌套锁（即一个同步块中又包含了其他同步块）的使用，并且避免锁操作（申请、释放）可能导致的死锁。编译器和处理器都必须遵守这些规则。

- 规则 7——临界区外（临界区前、临界区后）的操作可以被重排到临界区之内。

规则 7 有点费解。需要从两个层次去理解。在编译（JIT 动态编译）的时候，编译器可能将临界区前、临界区后的语句移到临界区之内，然后再在临界区的开始之前和结束之后相应地插入获取屏障和释放屏障。这些由编译器插入的内存屏障会得到处理器的“尊重”——处理器不会再将这被重排到临界区内的语句（对应的指令）重排序到临界区之外（规则 1）。也就是说，Java 源代码中的临界区外的语句可以被重排到临界区之内。但是，JIT 动态编译（从字节码到机器码的编译）过后的目标代码中的临界区之外的指令，由于编译器插入的内存屏障的作用无法被重排到临界区之内¹²。

规则 1、规则 4、规则 5 和规则 6 是保障锁的作用所必需的，其他规则可能与具体的 Java 虚拟机有关。这些规则其实是从 Java 语言（高级语言）的角度描述的，因此编译器能够遵守这些规则是因为它“认识”Java 语言。而处理器仅“认识”指令，因此处理器要遵守上述规则就需要借助相关指令（即内存屏障）。

¹² 以 OpenJDK 在 x86 处理器上的实现为例，Java 虚拟机在获得锁和释放锁的时候使用了 x86 的 Lock 前缀指令，该指令能够禁止其前或者其后的读写操作与其之间的重排序，这使得动态编译后的目标代码中的临界区外的操作不会被重排序到临界区之内。

3.8 轻量级同步机制：volatile 关键字

volatile 字面有“易挥发”的意思，引申开来就是有“不稳定”的意思。volatile 关键字用于修饰共享可变变量，即没有使用 final 关键字修饰的实例变量或静态变量，相应的变量就被称为 volatile 变量，如下所示：

```
private volatile int logLevel;
```

volatile 关键字表示被修饰的变量的值容易变化（即被其他线程更改），因而不稳定。volatile 变量的不稳定性意味着对这种变量的读和写操作都必须从高速缓存或者主内存（也是通过高速缓存读取）中读取，以读取变量的相对新值。因此，volatile 变量不会被编译器分配到寄存器进行存储，对 volatile 变量的读写操作都是内存访问（访问高速缓存相当于主内存）操作。

volatile 关键字常被称为轻量级锁，其作用与锁的作用有相同的地方：保证可见性和有序性。所不同的是，在原子性方面它仅能保障写 volatile 变量操作的原子性，但没有锁的排他性；其次，volatile 关键字的使用不会引起上下文切换（这是 volatile 被冠以“轻量级”的原因）。因此，volatile 更像是一个轻量级简易（功能比锁有限）锁。

3.8.1 volatile 的作用

volatile 关键字的作用包括：保障可见性、保障有序性和保障 long/double 型变量读写操作的原子性。

约定

访问同一个 volatile 变量的线程被称为同步在这个变量之上的线程，其中读取这个变量的线程被称为读线程，更新这个变量的线程被称为写线程。一个线程可以既是读线程又是写线程。

volatile 关键字能够保障对 long/double 型变量的写操作具有原子性。在 Java 语言中，对 long 型和 double 型以外的任何类型的变量的写操作都是原子操作。考虑到某些 32 位 Java 虚拟机上对 long/double 型变量进行的写操作可能不具有原子性，正如第 2 章的“long/double 型变量写操作的非原子 Demo”（见清单 2-6）所展示的，Java 语言规范特别地规定对 long/double 型 volatile 变量的写操作和读操作也具有原子性。因此，要解决上述 Demo 中出现的问题（读取到更新的“中间结果”），我们只需要将其中的共享变量 value 采用 volatile 修饰，如下所示：

```
static volatile long value = 0;
```


但是，`volatile` 仅仅保障对其修饰的变量的写操作（以及读操作）本身的原子性，而这并不表示对 `volatile` 变量的赋值操作一定具有原子性。例如，如下对 `volatile` 变量 `count1` 的赋值操作并不是原子操作：

```
count1 = count2 + 1;
```

如果变量 `count2` 也是一个共享变量，那么该赋值操作实际上是一个 `read-modify-write` 操作。其执行过程中其他线程可能已经更新了 `count2` 的值，因此该操作不具备不可分割性，也就不是原子操作。如果变量 `count2` 是一个局部变量，那么该赋值操作就是一个原子操作。

一般而言，对 `volatile` 变量的赋值操作，其右边表达式中只要涉及共享变量（包括被赋值的 `volatile` 变量本身），那么这个赋值操作就不是原子操作。要保障这样操作的原子性，我们仍然需要借助锁。

又如这样一个赋值操作：

```
volatile Map aMap = new HashMap();
```

可以分解为如下伪代码所示的几个子操作：

```
objRef = allocate(HashMap.class); // 子操作①：分配对象所需的存储空间
invokeConstructor(objRef); // 子操作②：初始化 objRef 引用的对象
aMap = objRef; // 子操作③：将对象引用写入变量 aMap
```

虽然 `volatile` 关键字仅保障其中的子操作③是一个原子操作，但是由于子操作①和子操作②仅涉及局部变量而未涉及共享变量，因此对变量 `aMap` 的赋值操作仍然是一个原子操作。

约定

`volatile` 关键字在原子性方面仅保障对被修饰的变量的读操作、写操作本身的原子性。如果要保障对 `volatile` 变量的赋值操作的原子性，那么这个赋值操作不能涉及任何共享变量（包括被赋值的 `volatile` 变量本身）的访问。

写线程对 `volatile` 变量的写操作会产生类似于释放锁的效果。读线程对 `volatile` 变量的读操作会产生类似于获得锁的效果。因此，`volatile` 具有保障有序性和可见性的作用。

对于 `volatile` 变量的写操作，Java 虚拟机会在该操作之前插入一个释放屏障，并在该操作之后插入一个存储屏障，如图 3-6 所示。

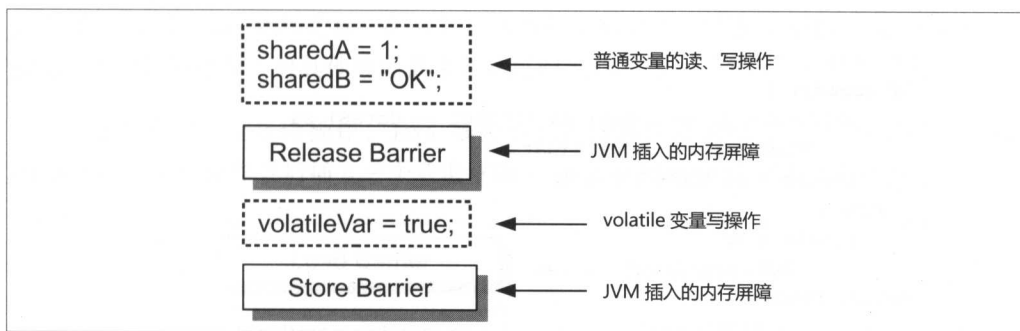


图 3-6 volatile 变量写操作与内存屏障

其中，释放屏障禁止了 volatile 写操作与该操作之前的任何读、写操作进行重排序，从而保证了 volatile 写操作之前的任何读、写操作会先于 volatile 写操作被提交，即其他线程看到写线程对 volatile 变量的更新时，写线程在更新 volatile 变量之前所执行的内存操作的结果对于读线程必然也是可见的。这就保障了读线程对写线程在更新 volatile 变量前对共享变量所执行的更新操作的感知顺序与相应的源代码顺序一致，即保障了有序性。

清单 3-6 展示了 volatile 对有序性的保障。该程序中的共享变量只有 ready 是 volatile 变量，其他共享变量都是普通的共享变量。

清单 3-6 volatile 对有序性的保障 Demo

```
@ConcurrencyTest(iterations = 200000)
public class VolatileOrderingDemo {
    private int dataA = 0;
    private long dataB = 0L;
    private String dataC = null;
    private volatile boolean ready = false;

    @Actor
    public void writer() {
        dataA = 1;
        dataB = 10000L;
        dataC = "Content...";
        ready = true;
    }

    @Observer({
        @Expect(desc = "Normal", expected = 1),
        @Expect(desc = "Impossible", expected = 2),
        @Expect(desc = "ready not true", expected = 3)
    })
    public int reader() {
```

```

        int result = 0;
        boolean allISOK;
        if (ready) {
            allISOK = (1 == dataA) && (10000L == dataB) &&
                "Content...".equals(dataC);
            result = allISOK ? 1 : 2;
        } else {
            result = 3;
        }
        return result;
    }

    public static void main(String[] args) throws InstantiationException,
        IllegalAccessException {

        // 调用测试工具运行测试代码
        TestRunner.runTest(VolatileOrderingDemo.class);
    }
}

```

上述程序运行后的输出类似如下：

```

=====2016-04-06 20:59:37 Wed=====

expected:1      occurrences:199999      ==>Normal

expected:2      occurrences:0      ==>Impossible

expected:3      occurrences:1      ==>ready not true

=====END=====

```

上述第 2 行情形的出现次数总是为 0。这说明当 reader 方法的执行线程读取到 ready 的值为 true 时，该线程所读取到的其他共享变量的值必然是 writer 方法的执行线程更新之后的值。这里，volatile 变量 ready 使得 reader 方法的执行线程对 writer 方法的执行线程所执行操作的感知顺序与源代码顺序一致。

volatile 虽然能够保障有序性，但是它不像锁那样具备排他性，所以并不能保障其他操作的原子性，而只能够保障对被修饰变量的写操作的原子性。因此，volatile 变量写操作之前的操作如果涉及共享可变量，那么竞态仍可能产生。这是因为共享变量被赋值给 volatile 变量的时候其他线程可能已经更新了该共享变量的值。

存储屏障具有冲刷处理器缓存的作用，因此在 volatile 变量写操作之后插入的一个存

存储屏障（参见图 3-6）就使得该存储屏障前所有操作的结果（包括 `volatile` 变量写操作及该操作之前的任何操作）对其他处理器来说是可同步的。

对于 `volatile` 变量读操作，Java 虚拟机会在该操作之前插入一个加载屏障（Load Barrier），并在该操作之后插入一个获取屏障（Acquire Barrier），如图 3-7 所示。

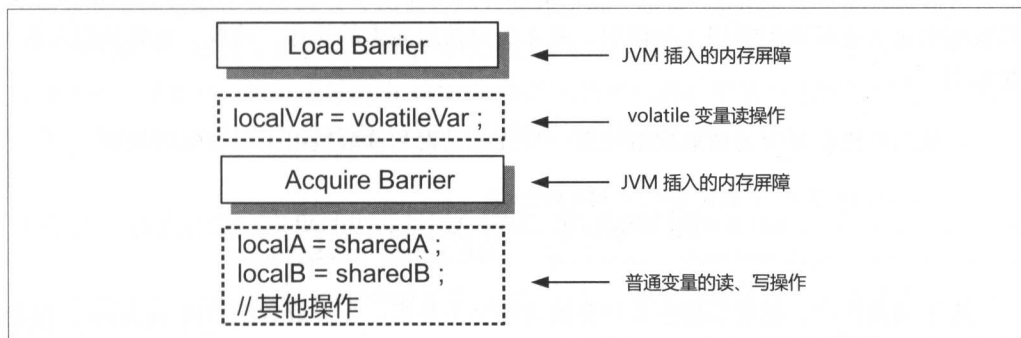


图 3-7 `volatile` 变量读操作与内存屏障

其中，加载屏障通过冲刷处理器缓存，使其执行线程（读线程）所在的处理器将其他处理器对共享变量（可能是多个变量）所做的更新同步到该处理器的高速缓存中。读线程执行的加载屏障和写线程执行的存储屏障配合在一起使得写线程对 `volatile` 变量的写操作以及在此之前所执行的其他内存操作的结果对读线程可见，即保障了可见性。因此，`volatile` 不仅仅保障了 `volatile` 变量本身的可见性，还保障了写线程在更新 `volatile` 变量之前执行的所有操作的结果对读线程可见。这种可见性保障类似于锁对可见性的保障，与锁不同的是 `volatile` 不具备排他性，因而它不能保障读线程读取到的这些共享变量的值是最新的，即读线程读取到这些共享变量的那一刻可能已经有其他写线程更新了这些共享变量的值。另外，获取屏障禁止了 `volatile` 读操作之后的任何读、写操作与 `volatile` 读操作进行重排序。因此它保障了 `volatile` 读操作之后的任何操作开始执行之前，写线程对相关共享变量（包括 `volatile` 变量和普通变量）的更新已经对当前线程可见。

另外，`volatile` 关键字也可以被看作给 JIT 编译器的一个提示，它相当于告诉 JIT 编译器相应变量的值可能被其他处理器更改，从而使 JIT 编译器不会对相应代码做出一些优化而导致可见性问题。例如，第 2 章我们讲到的可见性问题实例（见清单 2-7）中的 `ready` 变量采用 `volatile` 修饰之后就可以避免可见性问题。

`volatile` 在有序性保障方面也可以从禁止重排序的角度理解，即 `volatile` 禁止了如下重排序：

- 写 `volatile` 变量操作与该操作之前的任何读、写操作不会被重排序；
- 读 `volatile` 变量操作与该操作之后的任何读、写操作不会被重排序。

综上所述，我们知道 `volatile` 关键字的作用体现在对其所修饰的变量的读、写操作上。

如果被修饰的变量是个数组，那么 `volatile` 关键字只能够对数组引用本身的操作（读取数组引用和更新数组引用）起作用，而无法对数组元素的操作（读取、更新数组元素）起作用。

对数组的操作可分为读取数组元素、写数组元素和读取数组引用这几种类型：

```
int i = anArray[0]; // 操作类型①：读取数组元素
anArray[1] = 1; // 操作类型②：写数组元素
volatile int[] anotherArray = anArray; // 操作类型③：读取数组引用
```

在上述操作中，类型①操作可以分解为两个子步骤：先读取数组引用 `anArray`，接着读取数组中的第 0 个元素。这里，第 1 个子步骤实际上是读取一个引用（相当于相应数组的内存地址，或者干脆理解为 C 语言中的指针），该子步骤是个 `volatile` 变量读取操作，它保障了当前线程能够读取到数组引用本身的相对新值；而第 2 个子步骤则是在指定的数组引用（内存地址）基础上计算偏移量来读取数组元素，它与 `volatile` 关键字没有关系。因此，它不能保障其读取到的值是相对新值。也就是说，在类型①操作中，`volatile` 关键字起到的作用是保障当前线程能够读取到的数组引用的相对新值，这个值仅代表相应数组的内存地址而已，而该操作所读取到的数组元素值是否是相对新值则无法通过 `volatile` 关键字保障。类似地，在类型②操作中，`volatile` 关键字起到的作用只是保障读取到的数组引用是一个相对新值，而对相应数组元素的写操作则没有可见性保障。类型③的操作是将一个数组的引用写入另外一个数组，这相当于更新另外一个数组的引用（内存地址），这里的赋值操作是能够触发 `volatile` 关键字的所有作用的。

如果要使对数组元素的读、写操作也能够触发 `volatile` 关键字的作用，那么我们可以使用本章的 3.10.2 节介绍类 `AtomicIntegerArray`、`AtomicLongArray` 和 `AtomicReferenceArray`。

类似地，对于引用型 `volatile` 变量，`volatile` 关键字只是保证读线程能够读取到一个指向对象的相对新的内存地址（引用），而这个内存地址指向的对象的实例/静态变量值是否是相对新的则没有保障。

注意

`volatile` 关键字在可见性方面仅仅是保证读线程能够读取到共享变量的相对新值。对于引用型变量和数组变量，`volatile` 关键字并不能保证读线程能够读取到相应对象的字段（实例变量、静态变量）、元素的相对新值。

3.8.2 volatile 变量的开销

volatile 变量的开销包括读变量和写变量两个方面。volatile 变量的读、写操作都不会导致上下文切换，因此 volatile 的开销比锁要小。写一个 volatile 变量会使该操作以及该操作之前的任何写操作的结果对其他处理器是可同步的，因此 volatile 变量写操作的成本介于普通变量的写操作和在临界区内进行的写操作之间。读取 volatile 变量的成本也比在临界区中读取变量要低（没有锁的申请与释放以及上下文切换的开销），但是其成本可能比读取普通变量要高一些。这是因为 volatile 变量的值每次都需要从高速缓存或者主内存中读取，而无法被暂存在寄存器中，从而无法发挥访问的高效性。

3.8.3 volatile 的典型应用场景与实战案例

volatile 除了用于保障 long/double 型变量的读、写操作的原子性，其典型使用场景还包括以下几个方面。

- **场景一** 使用 volatile 变量作为状态标志。在该场景中，应用程序的某个状态由一个线程设置，其他线程会读取该状态并以该状态作为其计算的依据（或者仅仅读取并输出这个状态值）。此时使用 volatile 变量作为同步机制的好处是一个线程能够“通知”另外一个线程某种事件（例如，网络连接断连之后重新连上）的发生，而这些线程又无须因此而使用锁，从而避免了锁的开销以及相关问题。
- **场景二** 使用 volatile 保障可见性。在该场景中，多个线程共享一个可变状态变量，其中一个线程更新了该变量之后，其他线程在无须加锁的情况下也能够看到该更新。
- **场景三** 使用 volatile 变量替代锁。volatile 关键字并非锁的替代品，但是在一定的条件下它比锁更合适（性能开销小、代码简单）。多个线程共享一组可变状态变量的时候，通常我们需要使用锁来保障对这些变量的更新操作的原子性，以避免产生数据不一致问题。利用 volatile 变量写操作具有的原子性，我们可以把这一组可变状态变量封装成一个对象，那么对这些状态变量的更新操作就可以通过创建一个新的对象并将该对象引用赋值给相应的引用型变量来实现。在这个过程中，volatile 保障了原子性和可见性，从而避免了锁的使用。

注意

volatile 关键字并非锁的替代品，volatile 关键字和锁各自有其适用条件。前者更适用于多个线程共享一个状态变量（对象），而后者更适用于多个线程共享一组状态变量。某些情形下，我们可以将多个线程共享的一组状态变量合并成一个对象，用一个 volatile 变量来引用该对象，从而使我们不必要使用锁。

- **场景四** 使用 `volatile` 实现简易版读写锁。在该场景中，读写锁是通过混合使用锁和 `volatile` 变量而实现的，其中锁用于保障共享变量写操作的原子性，`volatile` 变量用于保障共享变量的可见性。因此，与 `ReentrantReadWriteLock` 所实现的读写锁不同的是，这种简易版读写锁仅涉及一个共享变量并且允许一个线程读取这个共享变量时其他线程可以更新该变量（这是因为读线程并没有加锁）。因此，这种读写锁允许读线程可以读取到共享变量的非最新值。该场景的一个典型例子是实现一个计数器，如清单 3-7 所示¹³。

清单 3-7 基于 `volatile` 的简易读写锁

```
public class Counter {  
    private volatile long count;  
    public long vaule() {  
        return count;  
    }  
    public void increment() {  
        synchronized (this) {  
            count++;  
        }  
    }  
}
```

下面我们通过某分布式系统的负载均衡模块的设计与实现这样的实战案例来进一步讲解上述应用场景。某分布式系统（以下简称为系统）在其业务处理过程中需要通过网络连接调用下游部件提供的服务，即发送请求给下游部件。下游部件是一个集群环境（即多台主机对外提供相同的服务）。因此，该系统调用其下游部件服务的时候需要进行负载均衡控制，即保障下游部件的各台主机上接收到的请求数分布均匀（统计意义上的均匀）。

该系统在调用其下游部件时的负载均衡控制需要在不重启应用程序、服务器的情况下满足以下几点要求。

- **要求 1** 需要支持多种负载均衡算法，如随机轮询算法和加权随机轮询算法等。
- **要求 2** 需要支持在系统运行过程中动态调整负载均衡算法，如从使用随机轮询算法调整为使用加权随机轮询算法。
- **要求 3** 在调用下游部件的过程中，下游部件中的非在线主机（如出现故障的主机）需要被排除在外，即发送给下游部件的请求不能被派发给非在线主机（因为那样会导致请求处理失败）。

¹³ 该代码中的 `increment` 方法从性能的角度来看，还可以进一步优化，本章后续内容会讲到这一点。

- 要求 4 下游部件的节点信息可动态调整，如出于维护的需要临时删除一个节点过后又将其重新添加回来。

这个负载均衡模块会涉及比较多的 `volatile` 的使用。该系统负责调用其下游部件服务的类为 `ServiceInvoker`，如清单 3-8 所示。

清单 3-8 `ServiceInvoker` 源码

```
public class ServiceInvoker {
    // 保存当前类的唯一实例
    private static final ServiceInvoker INSTANCE = new ServiceInvoker();
    // 负载均衡器实例，使用 volatile 变量保障可见性
    private volatile LoadBalancer loadBalancer;

    // 私有构造器
    private ServiceInvoker() {
        // 什么也不做
    }

    /**
     * 获取当前类的唯一实例
     */
    public static ServiceInvoker getInstance() {
        return INSTANCE;
    }

    /**
     * 根据指定的负载均衡器派发请求到特定的下游部件
     *
     * @param request
     *        待派发的请求
     */
    public void dispatchRequest(Request request) {
        // 这里读取 volatile 变量 loadBalancer
        Endpoint endpoint = getLoadBalancer().nextEndpoint();

        if (null == endpoint) {
            // 省略其他代码

            return;
        }

        // 将请求发给下游部件
        dispatchToDownstream(request, endpoint);
    }
}
```



```
// 真正将指定的请求派发给下游部件
private void dispatchToDownstream(Request request, Endpoint endpoint) {
    Debug.info("Dispatch request to " + endpoint + ":" + request);
    // 省略其他代码
}

public LoadBalancer getLoadBalancer() {
    // 读取负载均衡器实例
    return loadBalancer;
}

public void setLoadBalancer(LoadBalancer loadBalancer) {
    // 设置或者更新负载均衡器实例
    this.loadBalancer = loadBalancer;
}
}
```

首先，我们使用 LoadBalancer 接口（源码见清单 3-9）对负载均衡算法进行抽象，并为系统支持的每个负载均衡算法创建一个 LoadBalancer 实现类，从而满足了要求 1。

清单 3-9 LoadBalancer 接口源码

```
public interface LoadBalancer {
    void updateCandidate(final Candidate candidate);
    Endpoint nextEndpoint();
}
```

接着，我们为 ServiceInvoker 设置一个实例变量 loadBalancer 用来保存 LoadBalancer 实例（即具体的负载均衡算法）。这里，我们使用 volatile 关键字修饰 loadBalancer，就是属于 volatile 关键字的场景二的运用：ServiceInvoker 的 dispatchRequest 方法会通过调用 getLoadBalancer()方法来读取 volatile 变量 loadBalancer，该方法运行在业务线程（即 Web 服务器的工作者线程）中。当系统的启动线程（即 main 线程）或者配置管理线程（负责配置数据的刷新）更新了变量 loadBalancer 的值之后，所有业务线程在无须使用锁的情况下也能够读取到更新后的 loadBalancer 变量值，这实现了对负载均衡算法的动态调整，即满足了要求 2。

再看看具体的负载均衡算法是如何满足要求 3 的。这个实现过程会涉及 volatile 关键字的场景一的运用。首先看加权轮询负载均衡算法的实现类 WeightedRoundRobinLoadBalancer，如清单 3-10 示。

清单 3-10 加权轮询负载均衡算法源码

```
/**
 * 加权轮询负载均衡算法实现类
```

```

*
* @author Viscent Huang
*/
public class WeightedRoundRobinLoadBalancer extends AbstractLoadBalancer {
    // 私有构造器
    private WeightedRoundRobinLoadBalancer(Candidate candidate) {
        super(candidate);
    }

    // 通过该静态方法创建该类的实例
    public static LoadBalancer newInstance(Candidate candidate)
        throws Exception {
        WeightedRoundRobinLoadBalancer lb =
            new WeightedRoundRobinLoadBalancer(candidate);
        lb.init();
        return lb;
    }

    // 在该方法中实现相应的负载均衡算法
    @Override
    public Endpoint nextEndpoint() {
        Endpoint selectedEndpoint = null;
        int subWeight = 0;
        int dynamicTotoalWeight;
        final double rawRnd = super.random.nextDouble();
        int rand;

        // 读取 volatile 变量 candidate
        final Candidate candiate = super.candidate;
        dynamicTotoalWeight = candiate.totalWeight;
        for (Endpoint endpoint : candiate) {
            // 选取节点以及计算总权重时跳过非在线节点
            if (!endpoint.isOnline()) {
                dynamicTotoalWeight -= endpoint.weight;
                continue;
            }
            rand = (int) (rawRnd * dynamicTotoalWeight);
            subWeight += endpoint.weight;
            if (rand <= subWeight) {
                selectedEndpoint = endpoint;
                break;
            }
        }
        return selectedEndpoint;
    }
}

```

WeightedRoundRobinLoadBalancer 在选取下游部件节点（Endpoint）的时候会先判断相应节点是否在线，它会跳过非在线的节点。再看看 Endpoint 类的源码（参见清单 3-11）。

清单 3-11 Endpoint 类源码

```
/**
 * 表示下游部件的节点
 * @author Viscent Huang
 */
public class Endpoint {
    public final String host;
    public final int port;
    public final int weight;
    private volatile boolean online = true;
    public Endpoint(String host, int port, int weight) {
        this.host = host;
        this.port = port;
        this.weight = weight;
    }
    public boolean isOnline() {
        return online;
    }
    public void setOnline(boolean online) {
        this.online = online;
    }
    // 完整代码见配套下载资源
}
```

这里 Endpoint 的 online 实例变量是个 volatile 变量，它用来表示相应节点的服务状态：是否在线。所有负载均衡算法实现类的抽象父类 AbstractLoadBalancer 内部会维护一个心跳线程（heartbeatThread）来定时检测下游部件各个节点的状态，并根据检测的结果来更新相应 Endpoint 的 online 实例变量，如清单 3-12 所示。这里心跳线程根据检测结果更新 volatile 变量 online 的值，而具体的负载均衡算法实现类（如 WeightedRoundRobinLoadBalancer）则根据变量 online 的值决定其动作（跳过还是不跳过相应节点，见清单 3-10），从而满足了要求 3。这个过程涉及了 volatile 关键字的场景一的运用。

清单 3-12 负载均衡算法抽象类 AbstractLoadBalancer 源码

```
/**
 * 负载均衡算法抽象实现类，所有负载均衡算法实现类的父类
 *
 * @author Viscent Huang
 */
public abstract class AbstractLoadBalancer implements LoadBalancer {
    private final static Logger LOGGER = Logger.getAnonymousLogger();
```

```

// 使用 volatile 变量替代锁 (有条件替代)
protected volatile Candidate candidate;
protected final Random random;
// 心跳线程
private Thread heartbeatThread;

public AbstractLoadBalancer(Candidate candidate) {
    if (null == candidate || 0 == candidate.getEndpointCount()) {
        throw new IllegalArgumentException("Invalid candidate " + candidate);
    }
    this.candidate = candidate;
    random = new Random();
}

public synchronized void init() throws Exception {
    if (null == heartbeatThread) {
        heartbeatThread = new Thread(new HeartbeatTask(), "LB_Heartbeat");
        heartbeatThread.setDaemon(true);
        heartbeatThread.start();
    }
}

@Override
public void updateCandidate(final Candidate candidate) {
    if (null == candidate || 0 == candidate.getEndpointCount()) {
        throw new IllegalArgumentException("Invalid candidate " + candidate);
    }
    // 更新 volatile 变量 candidate
    this.candidate = candidate;
}

/*
 * 留给子类实现的抽象方法
 */
* @see io.github.viscent.mtia.ch3.volatilecase.LoadBalancer#nextEndpoint()
*/
@Override
public abstract Endpoint nextEndpoint();

protected void monitorEndpoints() {
    // 读取 volatile 变量
    final Candidate currCandidate = candidate;
    boolean isTheEndpointOnline;

    // 检测下游部件状态是否正常
    for (Endpoint endpoint : currCandidate) {
        isTheEndpointOnline = endpoint.isOnline();
        if (doDetect(endpoint) != isTheEndpointOnline) {

```

```

        endpoint.setOnline(!isTheEndpointOnline);
        // 省略记录日志的代码
    }
} // for 循环结束
}

// 检测指定的节点是否在线
private boolean doDetect(Endpoint endpoint) {
    // ...
}

private class HeartbeatTask implements Runnable {
    @Override
    public void run() {
        try {
            while (true) {
                // 检测节点列表中的所有节点是否在线
                monitorEndpoints();
                Thread.sleep(2000);
            }
        } catch (InterruptedException e) {
            // 什么也不做
        }
    }
} // HeartbeatTask 类结束
}

```

再看看要求 4 的满足与 `volatile` 关键字的场景三的运用之间的关系。从 `WeightedRoundRobinLoadBalancer` 的源码（见清单 3-10）可以看出，负载均衡算法的 `nextEndpoint` 方法选取下游部件节点的时候会用到一个关键的 `volatile` 实例变量 `candidate`，该变量由负载均衡算法的抽象父类 `AbstractBalancer`（见清单 3-12）定义，其类型为 `Candidate`（见清单 3-13）。

清单 3-13 Candidate 类源码

```

public final class Candidate implements Iterable<Endpoint> {
    // 下游部件节点列表
    private final Set<Endpoint> endpoints;
    // 下游部件节点的总权重
    public final int totalWeight;

    public Candidate(Set<Endpoint> endpoints) {
        int sum = 0;
        for (Endpoint endpoint : endpoints) {
            sum += endpoint.weight;
        }
    }
}

```

```

totalWeight = sum;
this.endpoints = endpoints;
}
// 完整代码见本书配套下载资源
}

```

Candidate 类包含了下游部件的节点列表 (endpoints) 以及列表中所有节点的总权重 (totalWeight)。这里的实例变量 totalWeight 作为一个冗余信息, 其作用是避免负载均衡算法每次都要计算总权重。如果我们要变更下游部件的节点信息 (如删除一个节点), 那么配置管理器 (一个单独的工作者线程) 只需要调用 AbstractBalancer (见清单 3-12) 子类的 updateCandidate 方法即可。updateCandidate 方法会直接更新 candidate 变量的值, 这里 volatile 保障了这个操作的原子性和可见性。这就是 volatile 关键字的场景三的运用。相反, 如果我们采用下面的设计:

```

public abstract class AbstractLoadBalancer implements LoadBalancer {
    // candidate 变量不可更新
    protected final Candidate candidate;
    public AbstractLoadBalancer(Candidate candidate) {
        this.candidate = candidate;
    }
    // 省略其他与清单 3-12 相同的代码
}

public class Candidate {
    // 使 endpoints 及 totalWeight 这两个实例变量可更改
    public Set<Endpoint> endpoints;
    public int totalWeight;
    // 省略其他与清单 3-13 相同的代码
}

```

那么, 变更下游部件节点信息的时候配置管理器需要更新 Candidate 实例 (候选节点信息) 的 endpoints 实例变量以及 totalWeight 实例变量的值, 如下伪代码所示:

```

endpoints = newEndpoints; // 操作①
totalWeight = calculateTotalWeight(newEndpoints); // 操作②

```

即便是先撇开可见性问题不谈, 我们也必须使上述操作成为原子操作, 否则这种更新可能导致数据不一致。当配置管理器 (线程) 执行完操作①而未执行完操作②的时候, 业务线程可能已经通过 nextEndpoint() 来读取 Candidate 实例的 endpoints 属性和 totalWeight 属性的值了, 此时这些业务线程读取到的 endpoints 属性是配置管理器更新过的新值, 而 totalWeight 属性的值则仍然是 endpoints 属性前一个值 (旧值) 计算出来的一个 (旧) 值, 即业务线程读取到的是一个错误的配置。因此, 我们并没有采用上述 (操作①和操作②) 方案来更新节点列表及相应的总权重, 而是巧妙地使用了 volatile 变量来保障这个更新操

作的原子性（以及可见性）。

3.9 实践：正确实现看似简单的单例模式

单例（Singleton）模式是 GOF（Gang of Four）设计模式中比较容易理解、运用也非常广泛的一个模式。但是实现一个能够在多线程环境下正常运作且能够兼顾到性能的有实用价值的单例类却不是一件容易的事情！正确实现该模式不仅仅具有实际工作上的意义，并且是对我们是否真正掌握锁以及 `volatile` 关键字这两种线程同步机制的一个检验：在多线程环境下正确实现单例模式要求我们对锁、`volatile` 变量以及可见性、原子性和有序性这些基础概念有准确而深入的理解！

单例模式所要实现的目标（效果）非常简单：保持一个类有且仅有一个实例¹⁴。出于性能的考虑，不少单例模式的实现会采用延迟加载（Lazy Loading）的方式，即仅在需要用到相应实例的时候才创建实例。从单线程应用程序的角度理解，采用延迟加载实现的一个单例模式如清单 3-14 所示。

清单 3-14 单线程版单例模式实现

```
public class SingleThreadedSingleton {
    // 保存该类的唯一实例
    private static SingleThreadedSingleton instance = null;

    // 省略实例变量声明
    /*
     * 私有构造器使其他类无法直接通过 new 创建该类的实例
     */
    private SingleThreadedSingleton() {
        // 什么也不做
    }

    /**
     * 创建并返回该类的唯一实例 <BR>
     * 即只有该方法被调用时该类的唯一实例才会被创建
     */
}
```

14 严格来说，所谓“一个类有且仅有一个实例”隐含着前提——这个类是一个 Java 虚拟机实例（进程）中的一个 Class Loader 所加载的类。这是考虑到了 Java 虚拟机的 Class Loader 机制：同一个类可以被多个 Class Loader 加载，这些 Class Loader 各自创建这个类的类实例（Class 本身也是个对象）。因此，如果有多个 Class Loader 加载同一个类，那么所谓“单例”就无法满足——这些 Class Loader 各自的类实例都创建该类的唯一一个实例，实际上被创建的实例数就等于加载这个类的 Class Loader 的数量。

```

    * @return
    */
    public static SingleThreadedSingleton getInstance() {
        if (null == instance) { // 操作①
            instance = new SingleThreadedSingleton(); // 操作②
        }
        return instance;
    }

    public void someService() {
        // 省略其他代码
    }
}

```

在多线程环境下，`getInstance()`中的 `if` 语句形成一个 `check-then-act` 操作，它不是一个原子操作。由于代码中未使用任何同步机制，因此该程序的运行可能出现线程交错的情形：在 `instance` 值还是 `null` 的时候，线程 T_1 和线程 T_2 同时执行到操作①。接着在 T_1 执行操作②前 T_2 已率先执行完操作②。下一时刻，当 T_1 执行到操作②的时候，尽管 `instance` 实际上已经不为 `null`，但是 T_1 此时依然会再创建一个实例（因为 T_1 执行操作①时 `instance` 为 `null`）。这就导致了多个实例的创建，从而违背了初衷。当然，我们不难想到通过加锁可以解决这种问题，代码如清单 3-15 所示。

清单 3-15 简单加锁实现的单例模式实现

```

public class SimpleMultithreadedSingleton {
    // 保存该类的唯一实例
    private static SimpleMultithreadedSingleton instance = null;

    /*
     * 私有构造器使其他类无法直接通过 new 创建该类的实例
     */
    private SimpleMultithreadedSingleton() {
        // 什么也不做
    }

    /**
     * 创建并返回该类的唯一实例 <BR>
     * 即只有该方法被调用时该类的唯一实例才会被创建
     *
     * @return
     */
    public static SimpleMultithreadedSingleton getInstance() {
        synchronized (SimpleMultithreadedSingleton.class) {
            if (null == instance) {
                instance = new SimpleMultithreadedSingleton();
            }
        }
    }
}

```



```

    }
    return instance;
}

public void someService() {
    // 省略其他代码
}
}

```

这种方法实现的单例模式固然是线程安全的，但是这意味着 `getInstance()` 的任何一个执行线程都需要申请锁。为了避免锁的开销，人们想到一个“聪明”的方法：在执行如清单 3-15 所示的临界区代码前先检查 `instance` 是否为 `null`；若 `instance` 不为 `null`，则 `getInstance()` 直接返回，否则才执行临界区。由于这种方法实现的 `getInstance()` 会两次检查 `instance` 的值是否为 `null`，因此它被称为双重检查锁定（Double-checked Locking, DCL）¹⁵，如清单 3-16 所示。

清单 3-16 基于双重检查锁定的错误单例模式实现

```

/**
 * 基于双重检查锁定的错误单例模式实现
 *
 * @author Viscent Huang
 */
public class IncorrectDCLSingleton {
    // 保存该类的唯一实例
    private static IncorrectDCLSingleton instance = null;

    /*
     * 私有构造器使其他类无法直接通过 new 创建该类的实例
     */
    private IncorrectDCLSingleton() {
        // 什么也不做
    }

    /**
     * 创建并返回该类的唯一实例 <BR>
     * 即只有该方法被调用时该类的唯一实例才会被创建
     *
     * @return
     */
}

```

15 双重检查锁定这种方法目前已经被视为反模式（Anti-Pattern），即不再提倡使用的方法。但是，不少现有系统和框架（如 Spring 框架）还在使用这种方法，因此掌握这种方法可能失效的原因以及正确实现的办法仍然具有实际意义。有关该方法的进一步信息参见：The "Double-Checked Locking is Broken" Declaration, <http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html>。

```

public static IncorrectDCLSingleton getInstance() {
    if (null == instance) { // 操作①: 第1次检查
        synchronized (IncorrectDCLSingleton.class) {
            if (null == instance) { // 操作②: 第2次检查
                instance = new IncorrectDCLSingleton(); // 操作③
            }
        }
    }
    return instance;
}

public void someService() {
    // 省略其他代码
}
}

```

尽管第1次检查(操作①)对变量 `instance` 的访问没有加锁从而使竞态仍然存在,但是乍一看,它似乎既避免了锁的开销又保障了线程安全:一个线程 T_1 执行到操作①的时候发现 `instance` 为 `null`,而此刻另外一个线程 T_2 可能恰好刚执行完操作③而使 `instance` 值不为 `null`;接着 T_1 获得锁而执行临界区代码的时候会再次判断 `instance` 值是否为 `null` (第2次检查),此时由于该线程是在临界区内读取共享变量 `instance` 的,因此 T_1 可以发现此刻 `instance` 值已经不为 `null`,于是, T_1 不会操作③(创建实例),从而避免了再次创建一个实例。当然,仅仅从可见性的角度分析结论确实如此。但是,在一些情形下为了确保线程安全光考虑可见性是不够的,我们还需要考虑重排序的因素。我们知道操作③可以分解为以下伪代码所示的几个独立子操作:

```

objRef = allocate(IncorrectDCLSingleton.class); // 子操作①: 分配对象所需的存储空间
invokeConstructor(objRef); // 子操作②: 初始化 objRef 引用的对象
instance = objRef; // 子操作③: 将对象引用写入共享变量

```

根据锁的重排序规则2和规则1(参见3.7节),临界区内的操作可以在临界区内被重排序。因此,JIT编译器可能将上述的子操作重排序为:子操作①→子操作③→子操作②,即在初始化对象之前将对象的引用写入实例变量 `instance` (正如我们在第2章清单2-10所示的Demo中所看到的现象)。由于锁对有序性的保障是有条件的(参见3.2.1节),而操作①(第1次检查)读取 `instance` 变量的时候并没有加锁,因此上述重排序对操作①的执行线程是有影响的:该线程可能看到一个未初始化(或未初始化完毕)的实例,即变量 `instance` 的值不为 `null`,但是该变量所引用的对象中的某些实例变量的变量值可能仍然是默认值,而不是构造器中设置的初始值。也就是说,一个线程在执行操作①的时候发现 `instance` 不为 `null`,于是该线程就直接返回这个 `instance` 变量所引用的实例,而这个实例可能是未初始化完毕的,这就可能导致程序出错!

在分析清楚问题的原因之后，解决方法也就不难想到：只需要将 `instance` 变量采用 `volatile` 修饰即可。这实际上是利用了 `volatile` 关键字的以下两个作用。

- 保障可见性：一个线程通过执行操作③修改了 `instance` 变量值，其他线程可以读取到相应的值（通过执行操作①）。
- 保障有序性：由于 `volatile` 能够禁止 `volatile` 变量写操作与该操作之前的任何读、写操作进行重排序，因此，用 `volatile` 修饰 `instance` 相当于禁止 JIT 编译器以及处理器将子操作②（对对象进行初始化的写操作）重排序到子操作③（将对象引用写入共享变量的写操作），这保障了一个线程读取到 `instance` 变量所引用的实例时该实例已经初始化完毕。

通过 `volatile` 关键字对上述两点的保障，双重检测锁定所要实现的效果才得以正确实现，如清单 3-17 所示。

清单 3-17 基于双重检查锁定的正确单例模式实现

```
public class DCLSingleton {
    /*
     * 保存该类的唯一实例，使用 volatile 关键字修饰 instance
     */
    private static volatile DCLSingleton instance;

    /*
     * 私有构造器使其他类无法直接通过 new 创建该类的实例
     */
    private DCLSingleton() {
        // 什么也不做
    }

    /**
     * 创建并返回该类的唯一实例 <BR>
     * 即只有该方法被调用时该类的唯一实例才会被创建
     *
     * @return
     */
    public static DCLSingleton getInstance() {
        if (null == instance) { // 操作①：第 1 次检查
            synchronized (DCLSingleton.class) {
                if (null == instance) { // 操作②：第 2 次检查
                    instance = new DCLSingleton(); // 操作③
                }
            }
        }
        return instance;
    }
}
```

```

public void someService() {
    // 省略其他代码
}
}

```

考虑到双重检测锁定法实现上容易出错,我们可以采用另外一种同样可以实现延迟加载的效果且比较简单的一种方法,如清单 3-18 所示。

清单 3-18 基于静态内部类的单例模式实现

```

public class StaticHolderSingleton {
    // 私有构造器
    private StaticHolderSingleton() {
        Debug.info("StaticHolderSingleton initied.");
    }

    private static class InstanceHolder {
        // 保存外部类的唯一实例
        final static StaticHolderSingleton INSTANCE = new StaticHolderSingleton();
    }

    public static StaticHolderSingleton getInstance() {
        Debug.info("getInstance invoked.");
        return InstanceHolder.INSTANCE;
    }

    public void someService() {
        Debug.info("someService invoked.");
        // 省略其他代码
    }

    public static void main(String[] args) {
        StaticHolderSingleton.getInstance().someService();
    }
}

```

我们知道类的静态变量被初次访问会触发 Java 虚拟机对该类进行初始化,即该类的静态变量的值会变为其初始值而不是默认值。因此,静态方法 `getInstance()` 被调用的时候 Java 虚拟机就会初始化这个方法所访问的内部静态类 `InstanceHolder`。这使得 `InstanceHolder` 的静态变量 `INSTANCE` 被初始化,从而使 `StaticHolderSingleton` 类的唯一实例得以创建。由于类的静态变量只会创建一次,因此 `StaticHolderSingleton` (单例类) 只会被创建一次。

正确实现延迟加载的单例模式还有一种更为简单的方法,那就是利用枚举 (Enum) 类型,如清单 3-19 所示。

清单 3-19 基于枚举类型的单例模式实现示例代码

```

public class EnumBasedSingletonExample {
    public static void main(String[] args) {
        Thread t = new Thread() {
            @Override
            public void run() {
                Debug.info(Singleton.class.getName());
                Singleton.INSTANCE.someService();
            };
            t.start();
        }

        public static enum Singleton {
            INSTANCE;
            // 私有构造器
            Singleton() {
                Debug.info("Singleton initd.");
            }

            public void someService() {
                Debug.info("someService invoked.");
                // 省略其他代码
            }
        }
    }
}

```

这里，枚举类型 Singleton 相当于一个单例类，其字段 INSTANCE 值相当于该类的唯一实例。这个实例是在 Singleton.INSTANCE 初次被引用的时候才被初始化的。仅访问 Singleton 本身（比如上述的 Singleton.class.getName()调用）并不会导致 Singleton 的唯一实例被初始化。

3.10 CAS 与原子变量

CAS (Compare and Swap) 是对一种处理器指令（例如 x86 处理器中的 cmpxchg 指令）的称呼。不少多线程相关的 Java 标准库类的实现最终都会借助 CAS。虽然在实际工作中多数情况下我们并不需要直接使用 CAS，但是理解 CAS 有助于我们更好地理解相关标准库类，以便恰当地使用它们。

3.10.1 CAS

前文（参见 3.8.3 节）我们讲到的一个简易读写锁的 `increment` 方法（参见清单 3-7）使用了一个内部锁来保障计数器自增这个操作的原子性：

```
public void increment() {
    synchronized (this) {
        count++;
    }
}
```

实际上，这里使用锁来保障原子性显得有点杀鸡用牛刀的样子！锁固然是功能最强大、适用范围也很广泛的同步机制，但是毕竟它的开销也是最大的。另外，`volatile` 虽然开销小一点，但是它无法保障“`count++`”这种自增操作的原子性（这也是我们在前文的代码中使用锁的一个原因）。事实上，保障像自增这种比较简单的操作的原子性我们有更好的选择——CAS。CAS 能够将 `read-modify-write` 和 `check-and-act` 之类的操作转换为原子操作。

我们知道“`count++`”（`count` 是共享变量）实际上是一个 `read-modify-write` 操作，它可以由 CAS 转换为一种一般性的 `if-then-act` 的操作，并由处理器保障该操作的原子性。这里，CAS 好比一个代理人（中间人），共享同一个变量 `V` 的多个线程就是它的客户。当客户需要更新变量 `V` 的值的时候，它们只需要请求（即调用）代理人代为修改，为此，客户要告诉代理人其看到的共享变量的当前值 `A` 及其期望的新值 `B`。CAS 作为代理人，相当于如下伪代码所示的函数：

```
boolean compareAndSwap(Variable V, Object A, Object B) {
    if (A == V.get()) { // check: 检查变量值是否被其他线程修改过
        V.set(B); // act: 更新变量值
        return true; // 更新成功
    }
    return false; // 变量值已被其他线程修改，更新失败
}
```

CAS 是一个原子的 `if-then-act` 的操作，其背后的假设是：当一个客户（线程）执行 CAS 操作的时候，如果变量 `V` 的当前值和客户请求（即调用）CAS 时所提供的变量值 `A`（即变量的旧值）是相等的，那么就说明其他线程并没有修改过变量 `V` 的值¹⁶。执行 CAS 时如果没有其他线程修改过变量 `V` 的值，那么下手最快的客户（当前线程）就会抢先将变量 `V` 的值更新为 `B`（新值），而其他客户（线程）的更新请求则会失败。这些失败的客户（线程）通常可以选择再次尝试，直到成功。这好比春节的时候抢购火车票，下手快的

¹⁶ 下文会讲到这种假设并不一定总是能够直接成立。

会抢先买到票，而下手慢的可以再次尝试，直到买到票。显然，这种更新机制是以 CAS 操作是一个原子操作为基础的，这一点直接由处理器来保障。

有了 CAS 以后，清单 3-7 中的计数器可以改写为清单 3-20 那样的代码。

清单 3-20 使用 CAS 实现线程安全的计数器

```
public class CASBasedCounter {
    private volatile long count;
    // 完整代码参见本书配套下载资源

    public long vaule() {
        return count;
    }

    public void increment() {
        long oldValue;
        long newValue;
        do {
            oldValue = count; // 读取共享变量的当前值
            newValue = oldValue + 1; // 计算共享变量的新值
        } while (!compareAndSwap(oldValue, newValue));
    }

    /*
     * 该方法是一个实例方法，且共享变量 count 是当前类的实例变量，因此这里没有必要在方法参数中声明
     * 一个表示共享变量的参数
     */
    private boolean compareAndSwap(long oldValue, long newValue) {
        // 完整代码参见本书配套下载资源
    }
}
```

上述 increment 方法中的 do-while 循环用于更新共享变量失败的时候继续重试，直到更新成功。这也是许多基于 CAS 的算法的代码模板（伪代码）：

```
do {
    oldValue = V.get(); // 读取共享变量 v 的旧值
    newValue = calculate(oldValue); // 计算变量 v 的新值
} while (!compareAndSwap(V, oldValue, newValue));
```

即在循环体中读取共享变量 V 的旧值（当前值）A，并以该值为输入经过一些列操作计算共享变量的新值 B，接着调用 CAS 试图将 V 的值更新为 B。若更新失败（说明更新期间其他线程修改了共享变量 V 的值）则继续重试，直到成功。

需要注意的是，CAS 只是保障了共享变量更新这个操作的原子性，它并不保障可见

性。因此，在上述代码中我们仍然采用 `volatile` 修饰共享变量 `count`。

上述代码中的 `compareAndSwap` 方法是利用 `java.util.concurrent.atomic.AtomicLongFieldUpdater` 类实现的，这只是便于我们实际运行代码。事实上，多数情况下我们会使用其他更加直接的工具类，它们是位于包 `java.util.concurrent.atomic` 下的被称为原子变量类的几个类。

注意 CAS 仅保障共享变量更新操作的原子性，它并不保障可见性。

3.10.2 原子操作工具：原子变量类

原子变量类 (Atomics) 是基于 CAS 实现的能够保障对共享变量进行 read-modify-write 更新操作的原子性和可见性的一组工具类。这里所谓的 read-modify-write 更新操作，是指对共享变量的更新不是一个简单的赋值操作，而是变量的新值依赖于变量的旧值，例如自增操作 “`count++`”。由于 `volatile` 无法保障自增操作的原子性，而原子变量类的内部实现通常借助一个 `volatile` 变量并保障对该变量的 read-modify-write 更新操作的原子性，因此它可以被看作增强型的 `volatile` 变量。原子变量类一共有 12 个，可以被分为 4 组，如表 3-2 所示。

表 3-2 原子变量类

分 组	类
基础数据型	AtomicInteger、AtomicLong、AtomicBoolean
数组型	AtomicIntegerArray、AtomicLongArray、AtomicReferenceArray
字段更新器	AtomicIntegerFieldUpdater、AtomicLongFieldUpdater、AtomicReferenceFieldUpdater
引用型	AtomicReference、AtomicStampedReference、AtomicMarkableReference

`AtomicLong` 类继承自 `Number` 类，它相当于清单 3-20 中所实现的计数器，其内部维护了一个 `long` 型 `volatile` 变量。`AtomicLong` 类对外暴露了相关方法用于实现针对该 `volatile` 变量的自增（自减）操作，这些操作是基于 CAS 实现的原子性操作。`AtomicLong` 类的常用方法如表 3-3 所示。

`AtomicLong` 类可以被看作一个增强型的 `volatile long` 变量：调用 `AtomicLong` 的 `get()` 方法相当于读取一个 `volatile` 变量；调用 `AtomicLong` 的 `incrementAndGet()` 等实现自增、自减的方法相当于写 `volatile` 变量，与直接写 `volatile` 变量所不同的是这些方法所执行的操作具有原子性。

表 3-3 AtomicLong 类的常用方法

方法声明	功 能	备 注
public final long get()	获取当前实例的当前数值	相当于读取一个 volatile 变量,因此该方法并不保障读线程读取到的数值是最新的
public final long getAndIncrement()	使当前实例的数值以原子操作的方式自增 1。该方法的返回值为自增前的数值	这些方法实现的操作都是原子操作,客户端代码无须在调用这些方法时加锁
public final long getAndDecrement()	使当前实例的数值以原子操作的方式自减 1。该方法的返回值为自减前的数值	
public final long incrementAndGet()	使当前实例的数值以原子操作的方式自增 1。该方法的返回值为自增后的数值	
public final long decrementAndGet()	使当前实例的数值以原子操作的方式自减 1。该方法的返回值为自减后的数值	
public final void set(long newValue)	设置当前实例的数值为指定的值	

下面通过一个实战案例以 AtomicLong 类为例来介绍基础数据型原子变量类的使用方法。某分布式系统的性能测试桩（Test Stub）需要记录其在测试过程中接收到的请求总数（Request Count）、处理成功数（Success Count）和处理失败数（Failure Count）这 3 个指标¹⁷，以便于测试后收集相关数据进行相互验证。因此，该测试桩记录这些指标的时候必须保障计数准确性为 100%。因此，我们必须保障测试桩程序的线程安全。考虑到计数是一个比较简单的操作，我们不希望为了保障正确性而引入锁。于是，原子变量类就是一个很好的选择。基于这种考虑，我们为测试桩设计了如清单 3-21 所示的计数器 Indicator 类。

清单 3-21 基于原子变量类的指标统计器

```
public class Indicator {  
    // 保存当前类的唯一实例  
    private static final Indicator INSTANCE = new Indicator();  
    /**
```

17 在测试桩响应中 HTTP 状态码为 2 开头（如 200）就算处理成功，否则都算处理失败。

```
* 记录请求总数
*/
private final AtomicLong requestCount = new AtomicLong(0);

/**
 * 记录处理成功总数
 */
private final AtomicLong successCount = new AtomicLong(0);

/**
 * 记录处理失败总数
 */
private final AtomicLong failureCount = new AtomicLong(0);

private Indicator() {
    // 什么也不做
}

// 返回该类的唯一实例
public static Indicator getInstance() {
    return INSTANCE;
}

public void newRequestReceived() {
    // 使总请求数增加1。 这里无须加锁
    requestCount.incrementAndGet();
}

public void newRequestProcessed() {
    // 使总请求数增加1。 这里无须加锁
    successCount.incrementAndGet();
}

public void requestProcessedFailed() {
    // 使总请求数增加1。 这里无须加锁
    failureCount.incrementAndGet();
}

public long getRequestCount() {
    return requestCount.get();
}

public long getSuccessCount() {
    return successCount.get();
}

public long getFailureCountCount() {
    return failureCount.get();
}
```

```

    }
    public void reset() {
        requestCount.set(0);
        successCount.set(0);
        failureCount.set(0);
    }
    // 完整代码见本书配套下载资源
}

```

这里我们直接使用 `AtomicLong` 作为计数器，在调用其相关方法的时候我们并不需要加锁。接着，我们会在 `Servlet Filter` 中调用 `Indicator` 实例的相关方法：测试桩每收到一个请求就将请求总数的值自增 1，每给其客户端一个成功响应就将处理成功数自增 1，每给其客户端一个失败的响应就将处理失败数自增 1，如清单 3-22 所示。

清单 3-22 在 `Servlet Filter` 中更新统计指标

```

public class CountingFilter implements Filter {
    final Indicator indicator = Indicator.getInstance();

    @Override
    public void doFilter(ServletRequest request, ServletResponse response,
        FilterChain chain) throws IOException, ServletException {
        indicator.newRequestReceived();
        StatusExposingResponse httpResponse = new StatusExposingResponse(
            (HttpServletResponse) response);
        chain.doFilter(request, httpResponse);

        int statusCode = httpResponse.getStatus();
        if (0 == statusCode || 2 == statusCode / 100) {
            indicator.newRequestProcessed();
        } else {
            indicator.requestProcessedFailed();
        }
    }
    // 完整代码见本书配套下载资源
}

```

尽管上面的 `Filter` 实例会被 Web 服务器的多个工作者线程并发调用，但是我们调用 `Indicator` 实例的相关方法时是直接调用的而没有加锁。

实现查看上述指标值的时候我们只需要调用 `Indicator` 的相应 `get` 方法，如清单 3-23 的 JSP 代码所示。

清单 3-23 查看统计指标值的 JSP 代码片段

```
Request Count:<%= Indicator.getInstance().getRequestCount() %><br>
```

```
Success Count:<%= Indicator.getInstance().getSuccessCount() %><br>
Failure Count:<%= Indicator.getInstance().getFailureCountCount() %><br>
```

Indicator 的 reset 方法调用了 3 个 AtomicLong 实例的 set() 方法将 3 个指标值清零, 以便再次测试。

AtomicInteger 的使用方法类似于 AtomicLong, 这里不再赘述。

AtomicBoolean 类乍一看似乎显得有些多余, 因为对布尔型变量的写操作本身就是个原子操作。实际上, 这里需要注意更新操作并不一定是简单地进行赋值。AtomicBoolean 类如同其他原子操作类一样, 它们是要实现以 read-modify-write 操作的原子性。下面通过一个实战案例来介绍 AtomicBoolean 的典型运用场景。某系统的告警 (Alarm) 模块中的类 AlarmMgr (告警管理器) 内部会维护一个工作者线程 (告警上报线程) 用于将告警信息上报 (发送) 到告警服务器。告警上报线程是在 AlarmMgr.init() 中创建并启动的, 为了避免该线程被重复创建 (即创建多个告警上报线程), 我们需要在 AlarmMgr.init() 进行相应的控制, 如清单 3-24 所示。

清单 3-24 AtomicBoolean 运用实例代码

```
public enum AlarmMgr implements Runnable {
    // 保存该类的唯一实例
    INSTANCE;
    private final AtomicBoolean initializing = new AtomicBoolean(false);
    boolean initInProgress;

    AlarmMgr() {
        // 什么也不做
    }

    public void init() {
        // 使用 AtomicBoolean 的 CAS 操作确保工作者线程只会被创建 (并启动) 一次
        if (initializing.compareAndSet(false, true)) {
            Debug.info("initializing...");
            // 创建并启动工作者线程
            new Thread(this).start();
        }
    }

    public int sendAlarm(String message) {
        int result = 0;
        // ...
        return result;
    }
}
```

```

@Override
public void run() {
    // ...
}
}

```

`AtomicBoolean` 变量 `initializing` 用于表示告警管理器初始化（即创建并启动告警上报线程）的状态。`initializing` 内部值为 `true` 表示正在初始化（或已初始化完毕），`false` 表示未开始初始化。`AlarmMgr.init()` 在创建（并启动）告警上报线程前会检查 `initializing` 的内部值：若 `initializing` 内部值为 `false`，则将其置为 `true` 以表示当前线程即将执行初始化；若 `initializing` 内部值为 `true`，则当前线程直接从 `AlarmMgr.init()` 返回。显然，在多线程环境下这个将 `initializing` 内部值从 `false` 调整为 `true` 的过程是一个 `check-then-act` 操作，若用锁来保障该操作的原子性，那么 `AlarmMgr.init()` 看起来会像这样：

```

public void init() {
    synchronized (this) {
        if (initInProgress) {
            return;
        }
        initInProgress = true;
    }
    Debug.info("initializing...");
    // 创建并启动工作者线程
    new Thread(this).start();
}

```

而实际上，我们使用了 `AtomicBoolean` 的 `compareAndSwap` 方法（相当于 CAS）来保障上述 `check-then-act` 操作的原子性，从而既避免了锁的开销，又使代码更加简单。

我们知道，即使采用 `volatile` 关键字修饰数组变量，也无法保障对相应元素的读、写操作的可见性和原子性。为此，Java 专门引入了 `AtomicIntegerArray`、`AtomicLongArray` 和 `AtomicReferenceArray` 这 3 个类。这几个类的使用方法与 `AtomicLong` 类似，只不过我们在调用这些类的相关原子操作方法时需要多指定一个数组下标。

`AtomicReference` 类和 `AtomicBoolean` 类比较类似，因为对引用型变量的写操作本身也是一个原子操作，这样看来 `AtomicReference` 类似乎显得多余。`AtomicReference` 类的主要功能可以理解是对引用型变量的有条件更新：更新引用变量时确保该变量的确是要修改的那个，即该变量没有被其他线程修改过。`AtomicReference` 类提供的相关方法针对的是引用型变量，而 `AtomicBoolean` 类提供的相关方法针对的是基础型变量。不过，如果我们把对象引用型变量的值看作一种特殊的值——表示内存地址的一个值，那么从理解上看二者针对的数据类型就没有实质性的区别了。

前面我们讲到 CAS 实现原子操作背后的一个假设是：共享变量的当前值与当前线程所提供的旧值相同，我们就认为这个变量没有被其他线程修改过。实际上，这个假设不一定总是成立，或者说它总是可以成立却是有条件的。例如，对于共享变量 V，当前线程看到它的值为 A 的那一刻，其他线程已经将其值更新为 B，接着在当前线程执行 CAS 的时候该变量的值又被其他线程更新为 A，那么此时我们是否认为变量 V 的值没有被其他线程更新过呢，或者说这种结果是否可以接受呢？这就是 ABA 问题，即共享变量的值经历了 A→B→A 的更新。ABA 问题是否可以接受或者可以容忍与要实现的算法有关，某些情形下我们无法容忍 ABA 问题。规避 ABA 问题也不难，那就是为共享变量的更新引入一个修订号（也称时间戳）。每次更新共享变量时相应的修订号的值就会被增加 1。也就是说，我们将共享变量 V 的值“扩展”成一个由变量实际值和相应的修订号所组成的元组（[共享变量实际值，修订号]）。于是，对于初始实际值为 A 的共享变量 V，它可能经历这样的变量更新：[A,0]→[B,1]→[A,1]。这里，虽然变量 V 的实际值仍然经历了 A→B→A 的更新，但是由于每次变量的更新都导致了相应修订号的增加，我们依然能够准确地判断究竟变量的值是否被其他线程修改过。AtomicStampedReference 类就是基于这种思想而产生的。

字段更新器（AtomicIntegerFieldUpdater、AtomicLongFieldUpdater、AtomicReferenceFieldUpdater）这 3 个类相对来说更加底层一点儿，可以将其理解为对 CAS 的一种封装，而原子变量类中的其他类都可以利用这几个类来实现。

3.11 对象的发布与逸出

我们知道线程安全问题产生的前提条件是多个线程共享变量。即使是 private 变量，它也可能被多个线程共享。例如如下代码：

```
public class Example {
    private Map<String, Integer> registry = new HashMap<String, Integer>();
    public void someService(String in) {
        // 访问 registry
    }
}
```

如果上述 someService 方法是被多个线程执行（比如多个线程的 run 方法调用了该方法），那么 private 变量 registry 实际上就是被多个线程共享。当然，这里我们可以使用锁、volatile 关键字来保障此情形下的线程安全。

多个线程共享变量还有其他途径，它们被统称为对象发布（Publish）。对象发布是指

使对象能够被其作用域之外的线程访问。常见的对象发布形式除了上述的共享 `private` 变量之外，还包括以下几种。

- 发布形式 1 将对象引用存储到 `public` 变量中。例如：

```
public Map<String, Integer> registry = new HashMap<String, Integer>();
```

从面向对象编程的角度来看，这种发布形式不太提倡，因为它违反了信息封装（`Information Hiding`）的原则，不利于问题定位。

- 发布形式 2 在非 `private` 方法（包括 `public`、`protected`、`package` 方法）中返回一个对象。例如：

```
private Map<String, Integer> registry = new HashMap<String, Integer>();
public Map<String, Integer> getRegistry(){
    return this.registry;
}
```

- 发布形式 3 创建内部类，使得当前对象（`this`）能够被这个内部类使用。例如：

```
public void startTask(final Object task) {
    Thread t = new Thread(new Runnable() {
        @Override
        public void run() {
            // 省略其他代码
        }
    });
    t.start();
}
```

上述代码中的“`new Runnable()`”所创建的匿名类可用访问其外层类的当前实例 `this`（通过“外层类名.`this`”这种语法访问），也就是说该匿名类的外层类发布了自身的当前实例。

- 发布形式 4 通过方法调用将对象传递给外部方法。

外部方法（`Alien Method`）指相对于某个类而言其他类的方法或者该类的可覆盖方法（即非 `private` 方法或者非 `final` 方法）。将一个对象传递给外部方法也会被视为对象发布。

因此，当我们需要发布一个对象的时候就需要注意与之相关的线程安全问题。当然，我们可以酌情使用锁、`volatile` 关键字来保障线程安全。下面我们将介绍其他能够保障线程安全的措施。

3.11.1 对象的初始化安全：重访 final 与 static

前面介绍的线程安全的单例模式的实现方法中静态内部类法（见清单 3-18）能够奏效的前提是静态变量只会被初始化一次。Java 中类的初始化实际上也采取了延迟加载的技术，即一个类被 Java 虚拟机加载之后，该类的所有静态变量的值都仍然是其默认值（引用型变量的默认值为 null，boolean 变量的默认值为 false），直到有个线程初次访问了该类的任意一个静态变量才使这个类被初始化——类的静态初始化块（“static{}”）被执行，类的所有静态变量被赋予初始值，如清单 3-25 所示。

清单 3-25 类的延迟初始化 Demo

```
public class ClassLazyInitDemo {
    public static void main(String[] args) {
        Debug.info(Collaborator.class.hashCode()); // 语句①
        Debug.info(Collaborator.number); // 语句②
        Debug.info(Collaborator.flag);
    }
    static class Collaborator {
        static int number = 1;
        static boolean flag = true;
        static {
            Debug.info("Collaborator initializing...");
        }
    }
}
```

上述 Demo 的运行输出类似如下：

```
[2016-09-10 20:52:44.419] [INFO] [main]:460141958
[2016-09-10 20:52:44.419] [INFO] [main]:Collaborator initializing...
[2016-09-10 20:52:44.420] [INFO] [main]:1
[2016-09-10 20:52:44.420] [INFO] [main]:true
```

可见，访问 Collaborator 类本身（语句①）仅仅使该类被 Java 虚拟机加载，而并没有使其被初始化（此时，从输出上看我们并没有看到 static 初始化块被调用）。从“Collaborator initializing...”在 number 的初始值 1 之前被输出可以看出，当一个线程（这里是 main 线程）初次访问类 Collaborator 的静态变量（语句②）时这个类才被初始化。

static 关键字在多线程环境下有其特殊的涵义，它能够保证一个线程即使在未使用其他同步机制的情况下也总是可以读取到一个类的静态变量的初始值（而不是默认值）。但是，这种可见性保障仅限于线程初次读取该变量。如果这个静态变量在相应类初始化完毕之后被其他线程更新过，那么一个线程要读取该变量的相对新值仍然需要借助锁、volatile 关

键字等同步机制。

在如清单 3-26 所示的代码中，init 方法所启动的线程至少可以看到语句①～语句③的操作结果，即该线程总是可以看到 static 字段 taskConfig 的初始值。如果 init 方法被执行的时候（甚至是在此之前）其他线程执行了 changeConfig 方法，那么 init 方法中启动的线程能否读取到 taskConfig 的相对新值也是没有保障的。这种情形下要保障可见性，我们仍然需要借助其他的线程同步机制。

清单 3-26 static 关键字可见性保障示例

```
public class StaticVisibilityExample {
    private static Map<String, String> taskConfig;
    static {
        Debug.info("The class being initialized...");
        taskConfig = new HashMap<String, String>(); // 语句①
        taskConfig.put("url", "https://github.com/Viscent"); // 语句②
        taskConfig.put("timeout", "1000"); // 语句③
    }

    public static void changeConfig(String url, int timeout) {
        taskConfig = new HashMap<String, String>(); // 语句④
        taskConfig.put("url", url); // 语句⑤
        taskConfig.put("timeout", String.valueOf(timeout)); // 语句⑥
    }

    public static void init() {
        // 该线程至少能够看到语句①～语句③的操作结果，而能否看到语句④～语句⑥的操作结果是没有保障的
        Thread t = new Thread() {
            @Override
            public void run() {
                String url = taskConfig.get("url");
                String timeout = taskConfig.get("timeout");
                doTask(url, Integer.valueOf(timeout));
            }
        };
        t.start();
    }

    private static void doTask(String url, int timeout) {
        // 省略其他代码

        // 模拟实际操作的耗时
        Tools.randomPause(500);
    }
}
```

对于引用型静态变量，`static` 关键字还能够保障一个线程读取到该变量的初始值时，这个值所指向（引用）的对象已经初始化完毕。

注意

`static` 关键字仅仅保障读线程能够读取到相应字段的初始值，而不是相对新值。

我们知道由于重排序的作用（参见清单 2-10），一个线程读取到一个对象引用时，该对象可能尚未初始化完毕，即这些线程可能读取到该对象字段的默认值而不是初始值（通过构造器或者初始化语句指定的值）。在多线程环境下 `final` 关键字有其特殊的作用：

当一个对象被发布到其他线程的时候，该对象的所有 `final` 字段（实例变量）都是初始化完毕的，即其他线程读取这些字段的时候所读取到的值都是相应字段的初始值（而不是默认值）。而非 `final` 字段没有这种保障，即这些线程读取该对象的非 `final` 字段时所读取到的值可能仍然是相应字段的默认值。对于引用型 `final` 字段，`final` 关键字还进一步确保该字段所引用的对象已经初始化完毕，即这些线程读取该字段所引用的对象的各个字段时所读取到的值都是相应字段的初始值。

假设两个线程分别执行清单 3-27 中的 `writer()` 和 `reader()`，那么 `reader()` 的执行线程读取到实例变量 `x` 的值一定为 1，而该线程读取到实例变量 `y` 的值则可能是 2（初始值）也可能是 0（默认值）。

清单 3-27 `final` 关键字的作用示例

```
public class FinalFieldExample {
    final int x;
    int y;
    static FinalFieldExample instance;

    public FinalFieldExample() {
        x = 1;
        y = 2;
    }

    public static void writer() {
        instance = new FinalFieldExample();
    }

    public static void reader() {
        final FinalFieldExample theInstance = instance;
        if (theInstance != null) {
            int diff = theInstance.y - theInstance.x;
            // diff 的值可能为 1 (=2-1)，也可能为 -1 (=0-1)
            print(diff);
        }
    }
}
```

```

    }

    private static void print(int x) {
        // ...
    }
}

```

在 JIT 编译器的内联 (Inline) 优化的作用下, `FinalFieldExample` 方法中的语句会被“挪入” `writer` 方法, 因此 `writer` 方法对应的指令可能被编译为与如下伪代码等效的代码:

```

objRef = allocate(FinalFieldExample.class); // 子操作①: 分配对象所需的存储空间
objRef.x = 1; // 子操作②: 对象初始化
objRef.y = 2; // 子操作③: 对象初始化
instance = objRef; // 子操作④: 将对象引用写入共享变量

```

其中, 子操作③ (非 `final` 字段初始化) 可能被 JIT 编译器、处理器重排序到子操作④ (对象发布) 之后, 因此当其他线程通过共享变量 `instance` 看到对象引用 `objRef` 的时候, 该对象的实例变量 `y` 可能还没有被初始化 (因为此时子操作③可能尚未被执行或者其结果尚未对其他处理器可见), 即这些线程看到的 `FinalFieldExample` 对象的 `y` 字段的值可能仍然是其默认值 0。而 `FinalFieldExample` 的字段 `x` 则是采用 `final` 关键字修饰, 因此 Java 虚拟机会将子操作② (final 字段初始化) 限定在子操作④前完成。这里所谓的限定是指 JIT 编译器不会将构造器中对 `final` 字段的赋值操作重排到子操作④之后, 并且还会禁止处理器做这种重排序¹⁸。通过这种限定, Java 虚拟机、处理器一起保障了对象 `instance` 被发布前其 `final` 字段 `x` 必然是初始化完毕的。

进一步, 对于引用型 `final` 字段, Java 语言规范还会保障其他线程看到包含该字段的对象时, 这个字段所引用的对象必然是初始化完毕的。如清单 3-28 所示, 当一个线程看到一个 `HTTPRangeRequest` 实例的时候, 该线程所看到的实例变量 `range` 所引用的对象必然是初始化完毕的, 但是该线程所看到的实例变量 `url` 的值可能仍然是 `null` (默认值)。

清单 3-28 `final` 关键字保障对象初始化完毕示例

```

public class HTTPRangeRequest {
    private final Range range;
    private String url;

    public HTTPRangeRequest(String url, int lowerBound, int upperBound) {
        this.url = url;
        this.range = new Range(lowerBound, upperBound);
    }
}

```

¹⁸ JIT 编译器是通过在编译后的机器码中插入特殊的内存屏障来实现这一点的。

```

public Range getRange() {
    return range;
}

public static class Range {
    private long lowerBound;
    private long upperBound;

    public Range(long lowerBound, long upperBound) {
        this.lowerBound = lowerBound;
        this.upperBound = upperBound;
    }
    // 完整代码见本书配套下载资源
}
}

```

我们知道，在 JIT 编译器的内联（Inline）优化的作用下，如下语句

```
instance = new HTTPRangeRequest("http://xyz.com/download/big.tar",0,1048576);
```

可能会被编译成与如下伪代码等效的指令：

```

objRef = allocate(HTTPRangeRequest.class); // 子操作①：分配对象所需的存储空间
objRef.url = "http://xyz.com/download/big.tar";
objRange = allocate(Range.class);
objRange.lowerBound = 0; // 子操作②：初始化对象 objRange
objRange.upperBound = 1048576; // 子操作③：初始化对象 objRange
objRef.range = objRange; // 子操作④：发布对象 objRange
instance = objRef; // 子操作⑤：发布对象 objRef

```

由于实例变量 `range`（引用型变量）采用 `final` 关键字修饰，因此 Java 语言会保障构造器中对该变量的初始化（赋值）操作（子操作④）以及该变量值所引用的对象（`Range` 实例）的初始化（子操作②和子操作③）被限定在子操作⑤前完成。这就保障了 `HTTPRangeRequest` 实例对外可见的时候，该实例的 `range` 字段所引用的对象已经初始化完毕。而 `url` 字段由于没有采用 `final` 修饰，因此 Java 虚拟机仍然可能将其重排序到子操作⑤之后。

在 3.8.3 节的实战案例中，我们使用 `final` 修饰 `AbstractLoadBalancer` 类（见清单 3-12）的 `java.util.Random` 型实例变量 `random`（随机数生成器），这不仅是因为该变量一经初始化就无须更新，更为重要的是由于我们要保障线程安全：`random` 变量的初始化是在一个线程（`main` 线程）进行中，而其使用（通过 `super.random.nextDouble()` 调用来生成随机数）是在另外一种线程（业务线程，即 `nextEndpoint()` 方法的执行线程）中进行的（参见清单 3-10），因此我们必须保障业务线程读取到 `random` 变量的值是初始值（而不是默认值

null)，并且该值所引用的 Random 实例是初始化完毕的。

这里需要注意，final 关键字只能保障有序性，即保障一个对象对外可见的时候该对象的 final 字段必然是初始化完毕的。final 关键字并不保障对象引用本身对外的可见性。

注意

当一个对象的引用对其他线程可见的时候，这些线程所看到的该对象的 final 字段必然是初始化完毕的。final 关键字的作用仅是这种有序性的保障，它并不能保障包含 final 字段的对象的引用自身对其他线程的可见性。

3.11.2 安全发布与逸出

安全发布就是指对象以一种线程安全的方式被发布。当一个对象的发布出现我们不期望的结果或者对象发布本身不是我们所期望的时候，我们就称该对象逸出（Escape）。逸出应该是我们要尽量避免的，因为它不是一种安全发布。

上述的发布形式 3（创建内部类，使得当前对象 this 能够被这个内部类使用）是最容易导致对象逸出的一种发布，它具体包括以下几种形式。

- 在构造器中将 this 赋值给一个共享变量。
- 在构造器中将 this 作为方法参数传递给其他方法。
- 在构造器中启动基于匿名类的线程。

由于构造器未执行结束意味着相应对象的初始化未完成，因此在构造器中将 this 关键字代表的当前对象发布到其他线程会导致这些线程看到的可能是一个未初始化完毕的对象，从而可能导致程序运行结果错误。

一般地，如果一个类需要创建自己的工作线程，那么我们可以为该类定义一个 init 方法（可以是 private 的），相应的工作线程可以在该方法或者该类的构造器创建，但是线程的启动则是在 init 方法中执行的。然后我们再为该类定义一个静态方法（工厂方法），该工厂方法会创建该类的实例并调用该实例的 init 方法，如清单 3-29 所示。

清单 3-29 在启动工作线程时实现对象安全发布范例

```
public class SafeObjPublishWhenStartingThread {
    private final Map<String, String> objectState;

    private SafeObjPublishWhenStartingThread(Map<String, String> objectState) {
        this.objectState = objectState;
        // 不在构造器中启动工作线程，以避免 this 逸出
    }
}
```

```

    }

    private void init() {
        // 创建并启动工作者线程
        new Thread() {
            @Override
            public void run() {
                // 访问外层类实例的状态变量
                String value = objectState.get("someKey");
                Debug.info(value);
                // 省略其他代码
            }
        }.start();
    }

    // 工厂方法
    public static SafeObjPublishWhenStartingThread newInstance(
        Map<String, String> objState) {
        SafeObjPublishWhenStartingThread instance =
            new SafeObjPublishWhenStartingThread(objState);
        instance.init();
        return instance;
    }
}

```

一个对象在其初始化过程中没有出现 `this` 逸出，我们就称该对象为正确创建的对象 (Properly Constructed Object)。要安全发布一个正确创建的对象，我们可以根据情况从以下几种方式中选择。

- 使用 `static` 关键字修饰引用该对象的变量。
- 使用 `final` 关键字修饰引用该对象的变量。
- 使用 `volatile` 关键字修饰引用该对象的变量。
- 使用 `AtomicReference` 来引用该对象。
- 对访问该对象的代码进行加锁。

3.12 本章小结

本章介绍了 Java 平台提供的各种线程同步机制。本章知识结构如图 3-8 所示。

Java 线程同步机制的幕后助手是内存屏障。不同同步机制的功能强弱不同，相应的开销以及可能导致的问题也不同，如表 3-4 所示。因此，我们需要根据实际情况选择一个功

能适用且开销较小的同步机制。

表 3-4 Java 线程同步机制的功能与开销/问题

	锁	volatile	CAS	final	static
原子性保障	具备	具备 ^②	具备	不涉及	不涉及
可见性保障	具备	具备	不具备	不具备	具备 ^③
有序性保障	具备	具备	不涉及	具备	具备 ^④
上下文切换?	可能 ^①	不会	不会	不会	可能 ^⑤
备注	① 被争用的锁可能导致上下文切换	② 仅能够保障对 volatile 变量读/写操作本身的原子性			③④ 仅在一个线程初次读取一个类的静态变量时起作用 ⑤ 静态变量所属类的初始化可能导致上下文切换

锁是 Java 平台中功能最强大的一种线程同步机制，同时其开销也最大，可能导致的问题也最多。被争用的锁会导致上下文切换，锁还可能导致死锁、锁死等线程活性故障。锁适用于存在多个线程对多个共享数据进行更新、check-then-act 操作或者 read-modify-write 操作这样的场景。

锁的排他性以及 Java 虚拟机在临界区前后插入的内存屏障使得临界区中的操作具有原子性。由此，锁还保障了写线程在临界区中执行操作在读线程看来是有序的，即保障了有序性。Java 虚拟机在 MonitorExit 对应的机器码后插入的内存屏障则保障了可见性。锁能够保障线程安全的前提是访问同一组共享数据的多个线程必须同步在同一个锁之上，否则原子性、可见性和有序性均无法得以保障。在满足貌似串行语义的前提下，临界区内以及临界区外的操作可以在各自范围内重排序。临界区外的操作可能会被 JIT 编译器重排到临界区内，但是临界区内的操作不会被编译器、处理器重排到临界区之外。

Java 中的所有锁都是可重入的。内部锁（synchronized）仅支持非公平锁，因此它可能导致饥饿。而显式锁（ReentrantLock）既支持非公平锁又支持公平锁，显式锁可能导致锁泄漏。内部锁和显式锁各有所长，各有所短。读写锁（ReadWriteLock）由于其内部实现的复杂性，仅适用于只读操作比更新操作要频繁得多且读线程持有锁的时间比较长的场景。读写锁（ReadWriteLock）中的读锁和写锁是一个锁实例所充当的两个角色，并不是两个独立的锁。

线程转储中可以包含锁的相关信息——线程在等待哪些锁，这些锁又是被哪些线程持有的。

`volatile` 相当于轻量级锁。在线程安全保障方面与锁相同的是，`volatile` 能够保障可见性、有序性；与锁不同的是 `volatile` 不具有排他性，也不会导致上下文切换。与锁类似，Java 虚拟机实现 `volatile` 对有序性和可见性的保障也是借助于内存屏障。从这个角度来看，`volatile` 变量写操作相当于释放锁，`volatile` 变量读操作相当于获得锁——Java 虚拟机通过在 `volatile` 变量写操作之前插入一个释放屏障，在 `volatile` 变量读操作之后插入一个获取屏障这种成对的释放屏障和获取屏障的使用实现了 `volatile` 对有序性的保障。类似地，Java 虚拟机在 `volatile` 变量写操作之后插入一个存储屏障，在 `volatile` 变量读操作之前插入一个加载屏障这种成对的存储屏障与加载屏障的使用实现了 `volatile` 对可见性的保障。

在原子性方面，`volatile` 仅能够保障 `long/double` 型变量写操作的原子性。如果要保障对 `volatile` 变量的赋值操作的线程安全，那么赋值操作右边的表达式不能涉及任何共享变量（包括被赋值的变量本身）。`volatile` 关键字在可见性、有序性和原子性方面的保障并不会对其修饰的数组的数组元素的读、写操作起作用。

`volatile` 变量写操作的成本介于普通变量的写操作和在临界区内进行的写操作之间。读取一个 `volatile` 变量总是意味着（通过高速缓存进行的）读内存操作，而不是从寄存器中读取。因此，`volatile` 变量读操作的成本比读取普通变量要略高一些，但比在临界区中读取变量要低。

`volatile` 的典型运用场景包括：一，使用 `volatile` 变量作为状态标志；二，使用 `volatile` 保障可见性；三，使用 `volatile` 变量替代锁；四，使用 `volatile` 实现简易版读写锁。

CAS 使得我们可以在不借助锁的情况下保障 `read-modify-write` 操作、`check-then-act` 操作的原子性，但是它并不保障可见性。原子变量类相当于基于 CAS 实现的增强型 `volatile` 变量（保障 `volatile` 无法保障的那一部分操作的原子性）。常用的原子变量类包括 `AtomicInteger`、`AtomicLong`、`AtomicBoolean` 等。`AtomicStampedReference` 则可以用于规避 CAS 的 ABA 问题。

`static` 关键字能够保证一个线程即使在未使用其他同步机制的情况下也总是可以读取到一个类的静态变量的初始值（而不是默认值）。对于引用型静态变量，`static` 还确保了该变量引用的对象已经初始化完毕。但是，`static` 的这种可见性和有序性保障仅在一个线程初次读取静态变量的时候起作用。

`final` 关键字在多线程环境下也有其特殊作用：当一个对象被发布到其他线程的时候，该对象的所有 `final` 字段（实例变量）都是初始化完毕的。而非 `final` 字段没有这种保障，即这些线程读取该对象的非 `final` 字段时所读取到的值可能仍然是相应字段的默认值。对于引用型 `final` 字段，`final` 关键字还进一步确保该字段所引用的对象已经初始化完毕。

实现对象的安全发布，通常可以依照以下顺序选择适用且开销最小的线程同步机制。

- 使用 `static` 关键字修饰引用该对象的变量。
- 使用 `final` 关键字修饰引用该对象的变量。
- 使用 `volatile` 关键字修饰引用该对象的变量。
- 使用 `AtomicReference` 来引用该对象。
- 对访问该对象的代码进行加锁。

为避免将 `this` 代表的当前对象逸出到其他线程，我们应该避免在构造器中启动工作者线程。通常我们可以定义一个 `init` 方法，在该方法中启动工作者线程。在此基础上，定义一个工厂方法来创建（并返回）相应的实例，并在该方法中调用该实例的 `init` 方法。

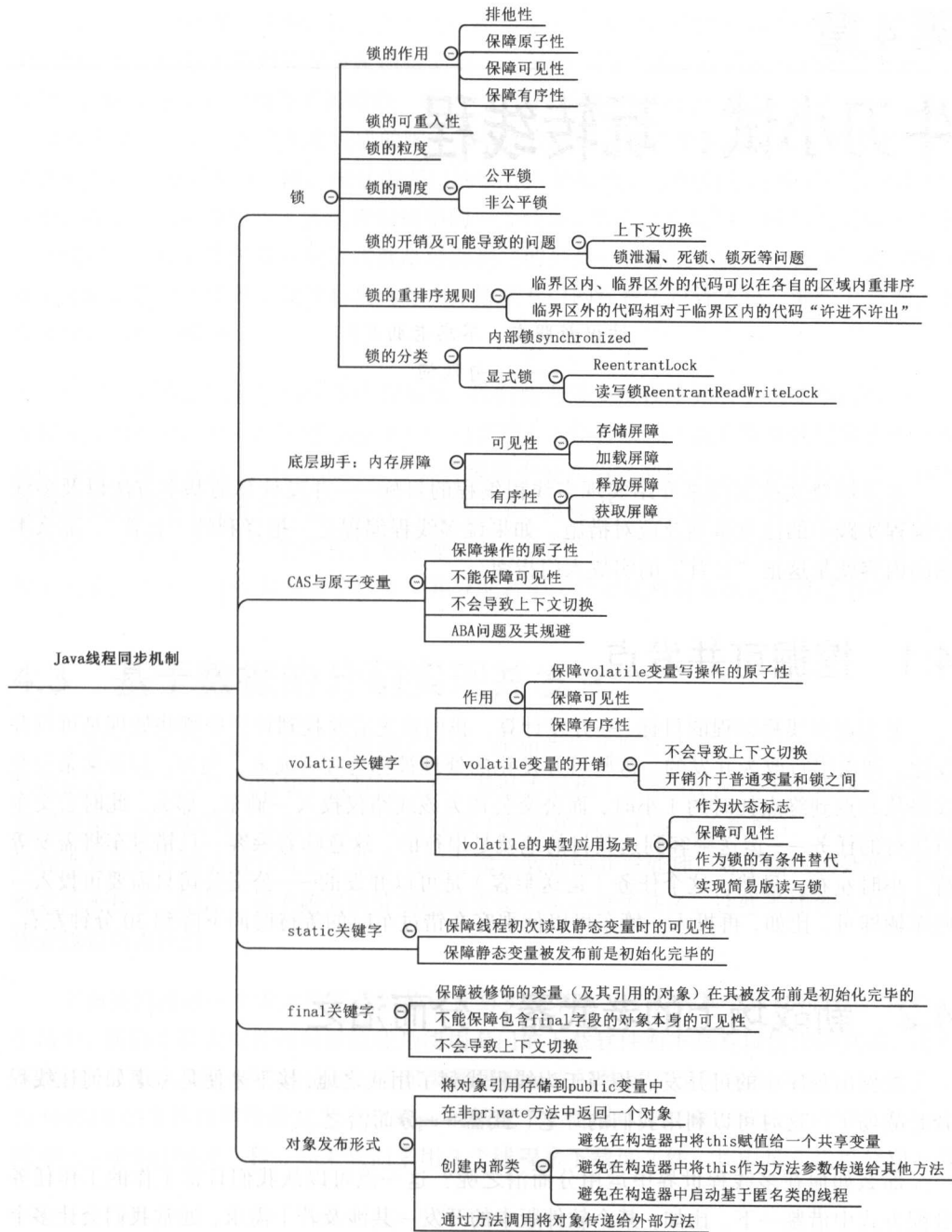


图 3-8 本章知识结构图

第4章

牛刀小试：玩转线程

结识新朋友，不忘老朋友！

——一句歌词

本章围绕实战案例来介绍实现多线程编程的目标——并发计算的基本方法以及多线程编程实践中的注意事项及应对措施。如果说多线程编程是一把锋利的“匕首”，那么本章的内容就是这把“匕首”的实战入门指南。

4.1 挖掘可并发点

要实现多线程编程的目标——并发计算，我们首先需要找到程序中哪些处理是可以并发化，即由串行改为并发的。这些可并发化的处理被称为可并发点。例如，假设某条公交线路从起点到终点耗时约1小时，而公交公司为该线路仅投入一辆车，那么，此时公交车所执行的任务——运送乘客到达其目的地就是串行的，这意味着乘客一旦错过车将需要等待1小时左右。显然，这个任务（运送乘客）是可以并发的——公交公司只需要再投入一些车辆即可。比如，再投入一辆车可以使乘客在错过车时的等待时间下降到30分钟左右。

4.2 新战场上的老武器：分而治之

挖掘出程序中的可并发点相当于为线程找到了用武之地，接下来便是考虑如何让线程奔赴战场了。这时可以利用我们的“老”武器——分而治之。

那么如何在多线程世界中运用分而治之呢？这一点可以从我们日常工作的工作任务分配方式中借鉴一下。比如，某个软件版本的开发一共涉及若干需求，通常我们会让多个开发人员各自负责其中的一个（或者多个）需求的开发。这种工作任务分配方式使得多个需求的开发能够以并发的方式进展。这里，将一批需求进行分解并指派到个人的过程就是

一个分而治之的过程。其中，每个开发人员相当于一个线程，其执行的任务（编写代码和单元测试等）的输入数据就是分配给他的需求。另外，一个需求的开发要经历需求分析、设计、编码、测试和验收等处理阶段。那么一个需求的这些处理阶段是从头到尾由一个人负责的还是分别由不同的人来完成的呢？这些处理阶段是可以由不同的人来完成的：需求分析由专门的分析人员来做；设计由专门的设计人员来做；编码由专门的编码人员来做。这里，每个处理阶段相当于多线程编程中的一个任务，执行这些处理阶段的人员就相当于一个线程。这种工作任务分配方式就是分而治之的另外一种体现：将一个任务（需求的开发）分解为若干子任务（设计和编码等）并指派专门的线程（设计人员、编码人员等）来负责执行这些子任务。

使用分而治之的思想进行多线程编程，我们首先需要将程序算法中只能串行的部分与可以并发的部分区分开来，然后使用专门的线程（工作者线程）去并发地执行那些可并发的部分（可并发点）。具体来说，多线程编程中分而治之的使用主要有两种方式：基于数据的分割和基于任务的分割。前者从数据入手，将程序的输入数据分解为若干规模较小的数据，并利用若干工作者线程并发处理这些分解后的数据。后者从程序的处理任务（步骤）入手，将任务分解为若干子任务，并分配若干工作者线程并发执行这些子任务。

4.3 基于数据的分割实现并发化

如果程序的原始输入数据的规模比较大，比如要从几百万条日志记录中统计出我们所需的信息，那么可以采用基于数据的分割。其基本思想（如图 4-1 所示）就是将原始输入数据按照一定的规则（比如均分）分解为若干规模较小的子输入（数据），并使用工作者线程来对这些子输入进行处理，从而实现输入数据的并发处理。对子输入的处理，我们称之为子任务。因此，基于数据的分割的结果是产生一批子任务，这些子任务由专门的工作者线程负责执行。

下面我们通过一个大文件下载器的设计与实现来进一步讲解基于数据的分割。在日常生活中，我们下载大文件的时候往往是使用专门的下载软件而不是直接使用浏览器。这些下载软件下载大文件时比较快的一个重要原因就是它们使用多线程技术。例如，一个大小为 600MB 的文件在网络带宽为 100Mbps 的情况下，使用单个线程下载该文件至少需要耗时 $48 (= 600 / (100 / 8))$ 秒。如果我们采用 3 个线程来下载该文件，其中每个线程分别下载

该文件的一个部分¹，那么下载这个文件所需的时间基本上可以减少为 $16(=600/3/(100/8))$ 秒，比起单线程下载节省了 $2/3$ 的时间。

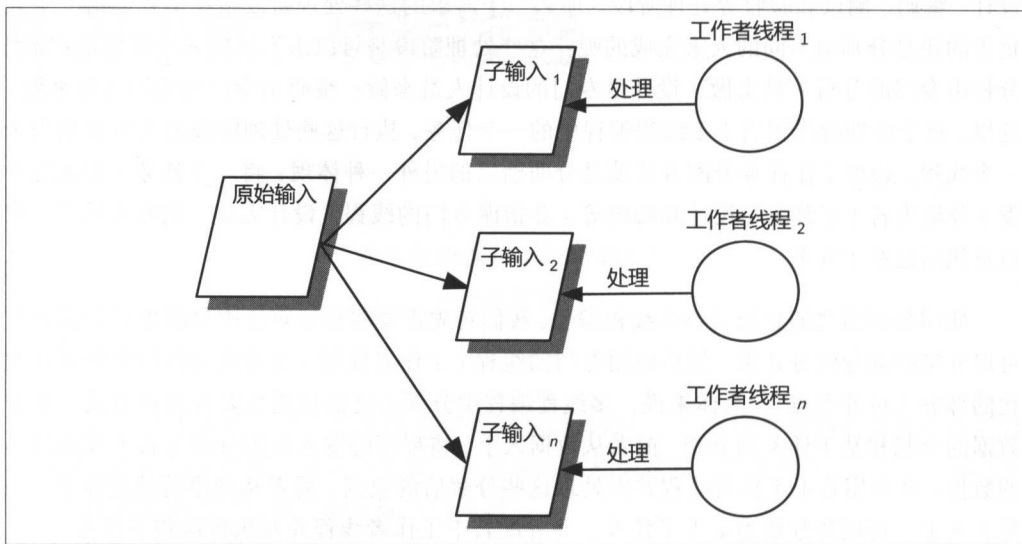


图 4-1 基于数据的分割并发化策略示意图

按照这个思路实现的一个基于多线程的大文件下载器，代码如清单 4-1 所示。首先，我们先获取待下载资源的大小²，这个大小相当于文件下载器的输入数据的原始规模（总规模）。接着，我们根据设定的下载线程数（workerThreadsCount）来决定子任务的总个数，并由此确定每个子任务负责下载的数据段的范围（起始字节到结束字节，lowerBound ~ upperBound）。然后我们分别创建相应的下载子任务（DownloadTask 类实例，代码见清单 4-2）并为每个下载任务创建相应的下载线程。这些线程启动后就会并发地下载大文件中的相应部分。

清单 4-1 大文件下载器入口类源码

```
/**
 * 大文件下载器
 *
```

- 1 即这些线程分别下载该文件第 $0 \sim 208666624 (=199 \times 1024 \times 1024)$ 字节部分、 $209715200 (=200 \times 1024 \times 1024) \sim 418381824 (=399 \times 1024 \times 1024)$ 字节部分和 $419430400 (=400 \times 1024 \times 1024) \sim 629145600 (=600 \times 1024 \times 1024)$ 字节部分。
- 2 通过给服务器发起一个 HTTP Head 请求，并读取响应中的 HTTP 头字段 Content-Length 就可以实现这一点。

```

* @author Viscent Huang
*/
public class BigFileDownloader {
    protected final URL requestURL;
    protected final long fileSize;
    /**
     * 负责已下载数据的存储
     */
    protected final Storage storage;
    protected final AtomicBoolean taskCanceled = new AtomicBoolean(false);

    public BigFileDownloader(String strURL) throws Exception {
        requestURL = new URL(strURL);

        // 获取待下载资源的大小（单位：字节）
        fileSize = retrieveFileSize(requestURL);
        Debug.info("file total size:%s", fileSize);
        String fileName = strURL.substring(strURL.lastIndexOf('/') + 1);
        // 创建负责存储已下载数据的对象
        storage = new Storage(fileSize, fileName);
    }

    /**
     * 下载指定的文件
     *
     * @param taskCount
     *         任务个数
     * @param reportInterval
     *         下载进度报告周期
     * @throws Exception
     */
    public void download(int taskCount, long reportInterval)
        throws Exception {

        long chunkSizePerThread = fileSize / taskCount;
        // 下载数据段的起始字节
        long lowerBound = 0;
        // 下载数据段的结束字节
        long upperBound = 0;
        DownloadTask dt;
        for (int i = taskCount - 1; i >= 0; i--) {
            lowerBound = i * chunkSizePerThread;
            if (i == taskCount - 1) {
                upperBound = fileSize;
            } else {
                upperBound = lowerBound + chunkSizePerThread - 1;
            }
        }
    }
}

```

```

        // 创建下载任务
        dt = new DownloadTask(lowerBound, upperBound, requestURL, storage,
            taskCanceled);
        dispatchWork(dt, i);
    }
    // 定时报告下载进度
    reportProgress(reportInterval);
    // 清理程序占用的资源
    doCleanup();
}

protected void doCleanup() {
    Tools.silentClose(storage);
}

protected void cancelDownload() {
    if (taskCanceled.compareAndSet(false, true)) {
        doCleanup();
    }
}

protected void dispatchWork(final DownloadTask dt, int workerIndex) {
    // 创建下载线程
    Thread workerThread = new Thread(new Runnable() {
        @Override
        public void run() {
            try {
                dt.run();
            } catch (Exception e) {
                e.printStackTrace();
                // 取消整个文件的下载
                cancelDownload();
            }
        }
    });
    workerThread.setName("downloader-" + workerIndex);
    workerThread.start();
}

// 根据指定的 URL 获取相应文件的大小
private static long retrieveFileSize(URL requestURL) throws Exception {
    // 完整代码见本书配套下载资源
}

// 报告下载进度
private void reportProgress(long reportInterval) throws InterruptedException {
    // 完整代码见本书配套下载资源
}
}

```

文件下载器的待下载资源相当于位于 Web 服务器上的一个大文件（输入），我们从逻辑上将其分解为若干子文件（起始字节和结束字节所表示的数据段），并使用多个工作者线程各自负责这些子文件的下载。比如，待下载资源的大小为 600MB，如果我们指定 3 个下载线程，那么每个下载线程只需要下载这个大文件中 200MB 的数据。因此，该案例实际上是将程序算法中从服务器上下载数据这个部分由原来单线程程序的串行处理变成了并发处理，即实现了并发化。

实战案例的启发

从上述案例中可以看出，基于数据的分割这种并发化策略是从程序处理的数据角度入手，将原始输入分解为若干规模更小的子输入，并将这些子输入指派给专门的工作者线程处理。基于数据的分割的结果是产生多个同质工作者线程，即任务处理逻辑相同的线程。例如，上述案例中的 BigFileDownloader（见清单 4-1）创建的工作者线程都是 DownloadTask（见清单 4-2）的实例。尽管基于数据的分割的基本思想不难理解，但是在实际运用中，我们往往有更多的细节需要考虑。

- 工作者线程数量的合理设置问题。在原始输入规模一定的情况下，增加工作者线程数量可以减小子输入的规模，从而减少每个工作者线程执行任务所需的时间。但是线程数量的增加也会导致其他开销（比如上下文切换）增加。例如，上述案例从表面上看，我们似乎可以指定更多的下载线程数来缩短资源下载耗时。比如，我们设定 10 个线程用于下载一个大小为 600MB 的资源，那么每个线程仅需要下载这个大文件中 60MB 的数据，这样看来似乎我们仅需要单线程下载的 1/6 时间就可以完成整个资源下载。但实际的结果却可能并非如此：增加下载线程数的确可以减少每个下载线程的输入规模（子输入的规模），从而缩短每个下载线程完成数据段下载所需的时间；但是这同时也增加了上下文切换的开销、线程创建与销毁的开销、建立网络连接的开销以及锁的争用等开销，而这些增加的开销可能无法被子输入规模减小所带来的好处所抵消。另一方面，工作者线程数量过少又可能导致子输入的规模仍然过大，这使得计算效率提升不明显。在本案例中，我们通过命令行参数指定工作者线程数量，本章后续内容会介绍工作者线程数的合理设置。
- 工作者线程的异常处理问题。对于一个工作者线程执行过程中出现的异常，我们该如何处理呢？例如，在本案例的一个下载线程执行过程中出现异常的时候，这个线程是可以进行重试（针对可恢复的故障）呢，还是说直接就算整个资源的下载失败呢？如果是算这个资源下载失败，那么此时其他工作者线程就没有必要继续运行下去了。因此，此时就涉及终止其他线程的运行问题。有关线程终止的内容，我们会在第 8 章详细介绍。

- 原始输入规模未知问题。在上述例子中，由于原始输入的规模是事先可知的，因此我们可以采用简单的均分对原始输入进行分解。但是，某些情况下我们可能无法事先确定原始输入的规模，或者事先确定原始输入规模是一个开销极大的计算。比如，要从几百个日志文件（其中每个文件可包含上万条记录）中统计出我们所需的信息，尽管理论上我们可以事先计算出总记录条数，但是这样做的开销会比较大，因而实际上这是不可行的。此时原始输入的规模就相当于事先不可知。对于这种原始输入规模事先不可知的问题，我们可以采用批处理的方式对原始输入进行分解：聚集了一批数据之后再将这些数据指派给工作者线程进行处理。这种方法类似于公安局办证中心办理护照的情形，虽然每天都可能有人去申请护照，但是办证中心并不是为每个申请人专门办理护照的，而是凑足一批申请人的材料后才进行统一办理的。在批处理的分解方式中，工作者线程往往是事先启动的，并且我们还需要考虑这些工作者线程的负载均衡问题，即新聚集的一批数据按照什么样的规则被指派给哪个工作者线程的问题。工作者线程的负载均衡问题类似于我们在第 3 章中举的负载均衡器的例子（参见清单 3-10）——如果我们把新聚集的一批数据看作一个请求，而把工作者线程看作一个“服务器节点”，那么这两个问题实际上就是一个问题。
- 程序的复杂性增加的问题。基于数据的分割产生的多线程程序可能比相应的单线程程序要复杂。例如，上述案例中虽然多个工作者线程并发地从服务器上下载大文件可以提升计算效率，但是它也带来一个问题：这些数据段是并发地从服务器上下载的，但是我们最终要得到的是一个完整的大文件，而不是几个较小的文件。因此，我们有两种选择：其中一种方法是，各个工作者线程将其下载的数据段分别写入各自的本地文件（子文件），等到所有工作者线程结束之后，我们再将这些子文件合并为我们最终需要的文件。显然，当待下载的资源非常大的时候合并这些子文件也是一笔不小的开销。另外一种方法是将各个工作者线程从服务器上下载到的数据都写入同一个本地文件，这个文件被写满之后就是我们最终所需的大文件。第二种方法看起来比较简单，但是这里面有个矛盾需要调和：文件数据是并发地从服务器上下载（读取）的，但是将这些数据写入本地文件的时候，我们又必须确保这些数据按照原始文件（服务器上的资源）的顺序被写入这个本地文件的相应位置（起始字节和结束字节）。另外，每个下载线程从网络读取一段数据（例如 1KB 的数据）就将其写入文件这种方法固然简单，但是容易增加 I/O 的次数。有鉴于此，上述案例我们采用了缓冲的方法：下载线程每次从网络读取的数据都是先被写入缓冲区（如清单 4-2 中加粗部分代码所示），只有当这个缓冲区满的时候其中的内容才会被写入本地文件（如清单 4-3 中加粗部分代码所示）。这个缓冲区是通过类 `DownloadBuffer`（代码参见清单 4-3）实现的，将缓冲区中的内容写入本地文件是通过类 `Storage`（代码参见清单 4-4）实现的。由此可见，上述案例中的多线程程序比起相应的单线程程序要复杂得多！

清单 4-2 下载子任务类 DownloadTask 源码

```

/**
 * 下载子任务
 *
 * @author Viscent Huang
 */
public class DownloadTask implements Runnable {
    private final long lowerBound;
    private final long upperBound;
    private final DownloadBuffer xbuf;
    private final URL requestURL;
    private final AtomicBoolean cancelFlag;

    public DownloadTask(long lowerBound, long upperBound, URL requestURL,
        Storage storage, AtomicBoolean cancelFlag) {
        this.lowerBound = lowerBound;
        this.upperBound = upperBound;
        this.requestURL = requestURL;
        this.xbuf = new DownloadBuffer(lowerBound, upperBound, storage);
        this.cancelFlag = cancelFlag;
    }

    // 对指定的 URL 发起 HTTP 分段下载请求
    private static InputStream issueRequest(URL requestURL, long lowerBound,
        long upperBound) throws IOException {
        // 完整代码见本书配套下载资源
    }

    @Override
    public void run() {
        if (cancelFlag.get()) {
            return;
        }
        ReadableByteChannel channel = null;
        try {
            channel = Channels.newChannel(issueRequest(requestURL, lowerBound,
                upperBound));
            ByteBuffer buf = ByteBuffer.allocate(1024);
            while (!cancelFlag.get() && channel.read(buf) > 0) {
                // 将从网络读取的数据写入缓冲区
                xbuf.write(buf);
                buf.clear();
            }
        } catch (Exception e) {
            throw new RuntimeException(e);
        } finally {

```

```
        Tools.silentClose(channel, xbuf);
    }
}
```

清单 4-3 缓冲区实现类 DownloadBuffer 源码

```
public class DownloadBuffer implements Closeable {
    /**
     * 当前 Buffer 中缓冲的数据相对于整个存储文件的位置偏移
     */
    private long globalOffset;
    private long upperBound;
    private int offset = 0;
    public final ByteBuffer byteBuf;
    private final Storage storage;

    public DownloadBuffer(long globalOffset, long upperBound, final Storage storage) {
        this.globalOffset = globalOffset;
        this.upperBound = upperBound;
        this.byteBuf = ByteBuffer.allocate(1024 * 1024);
        this.storage = storage;
    }

    public void write(ByteBuffer buf) throws IOException {
        int length = buf.position();
        final int capacity = byteBuf.capacity();

        // 当前缓冲区已满，或者剩余容量不够容纳新数据
        if ((offset + length) > capacity || length == capacity) {
            // 将缓冲区中的数据写入文件
            flush();
        }
        byteBuf.position(offset);
        buf.flip();
        byteBuf.put(buf);
        offset += length;
    }

    public void flush() throws IOException {
        int length;
        byteBuf.flip();
        length = storage.store(globalOffset, byteBuf);
        byteBuf.clear();
        globalOffset += length;
        offset = 0;
    }
}
```

```

public void close() throws IOException {
    if (globalOffset < upperBound) {
        flush();
    }
}
}
}

```

清单 4-4 文件存储实现类 Storage 源码

```

public class Storage implements Closeable, AutoCloseable {
    private final RandomAccessFile storeFile;
    private final FileChannel storeChannel;
    protected final AtomicLong totalWrites = new AtomicLong(0);

    public Storage(long fileSize, String fileShortName) throws IOException {
        String fullFileName = System.getProperty("java.io.tmpdir") + "/"
            + fileShortName;
        String localFileName;
        localFileName = createStoreFile(fileSize, fullFileName);
        storeFile = new RandomAccessFile(localFileName, "rw");
        storeChannel = storeFile.getChannel();
    }

    /**
     * 将 data 中指定的数据写入文件
     *
     * @param offset
     *      写入数据在整个文件中的起始偏移位置
     * @param byteBuf
     *      byteBuf 必须在该方法调用前执行 byteBuf.flip()
     * @throws IOException
     * @return 写入文件的数据长度
     */
    public int store(long offset, ByteBuffer byteBuf)
        throws IOException {
        int length;
        storeChannel.write(byteBuf, offset);
        length = byteBuf.limit();
        totalWrites.addAndGet(length);
        return length;
    }

    public long getTotalWrites() {
        return totalWrites.get();
    }

    private String createStoreFile(final long fileSize, String fullFileName)
        throws IOException {

```

```

File file = new File(fullFileName);
Debug.info("create local file:%s", fullFileName);
RandomAccessFile raf;
raf = new RandomAccessFile(file, "rw");
try {
    raf.setLength(fileSize);
} finally {
    Tools.silentClose(raf);
}
return fullFileName;
}

@Override
public synchronized void close() throws IOException {
    if (storeChannel.isOpen()) {
        Tools.silentClose(storeChannel, storeFile);
    }
}
}

```

4.4 基于任务的分割实现并发化

为了提高任务的执行效率，我们可能使用多个线程去共同完成一个任务的执行。这就是基于任务的分割，其基本思想（如图 4-2 所示）是将任务（原始任务）按照一定的规则分解成若干子任务，并使用专门的工作者线程去执行这些子任务，从而实现任务的并发执行。这种思想在日常生活中也常有体现。比如，饭店为了完成给顾客提供餐饮服务这个原始任务，需要完成这么几个子任务：接待客人、安排桌位、点餐、烹煮菜肴、上菜、收款等。显然，没有哪家饭店会让一个服务员从头至尾地为每位顾客去完成这几个子任务。饭店为了提高其接待能力（吞吐率）会安排专门的服务员（迎宾、点餐员、传菜员、厨师以及收银员等）分别负责这几个子任务的执行。这里，饭店其实是把原始任务（为客人提供餐饮服务）分解为若干子任务，并指派专门的工作人员（工作者线程）去执行这些子任务。

按照原始任务的分解方式来划分，基于任务的分解可以分为按任务的资源消耗属性分割和按处理步骤分割这两种。

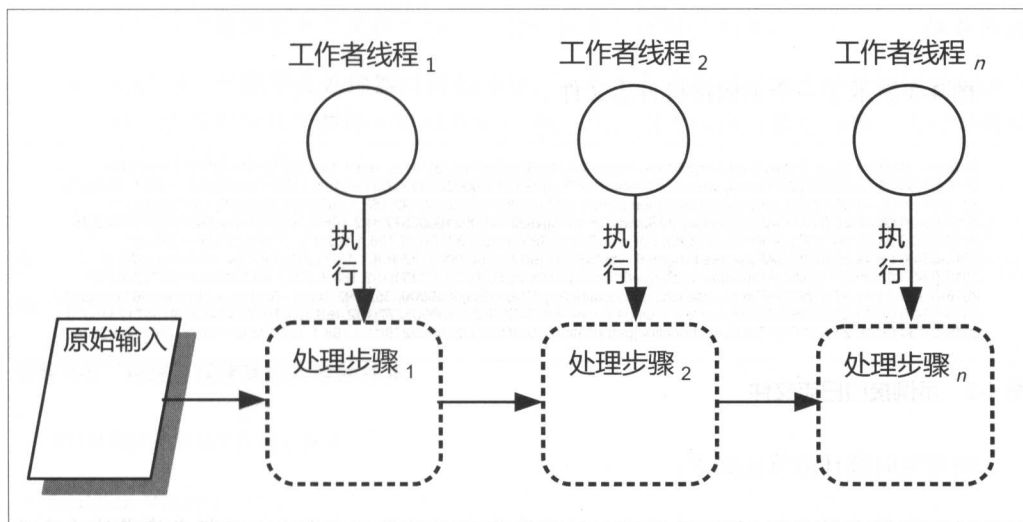


图 4-2 基于任务的分割并发化策略示意图

4.4.1 按任务的资源消耗属性分割

线程所执行的任务按照其消耗的主要资源可划分为 CPU 密集型 (CPU-intensive) 任务和 I/O 密集型 (I/O-intensive) 任务。执行这些任务的线程也相应地被称为 CPU 密集型线程和 I/O 密集型线程。CPU 密集型任务执行过程中消耗的主要资源是 CPU 时间，CPU 密集型任务的一个典型例子是加密和解密；I/O 密集型任务执行过程中消耗的主要资源是 I/O 资源（如网络 and 磁盘等），典型的 I/O 密集型任务包括文件读写、网络读写等。对任务的这种划分有点类似于我们将影视戏剧作品中的人物划分为“好人”和“坏人”。而在现实世界中我们往往很难找到一个纯粹的“好人”和纯粹的“坏人”。类似地，一个线程所执行的任务实际上往往同时兼具 CPU 密集型任务和 I/O 密集型任务特征，我们称之为混合型任务。有时候，我们可能需要将这种混合型任务进一步分解为 CPU 密集型和 I/O 密集型这两种子任务，并使用专门的工作者线程来负责这些子任务的执行，以提高并发性。

下面看一个实战案例。某分布式电信系统（以下简称“该系统”）需要一款统计工具，用于从指定的接口日志文件中统计出外部系统（部件）处理指定请求（操作）的响应延时（即从该系统给指定外部系统发送请求到该系统接收到相应响应之间的时间差）情况。接口日志文件格式如下：

```
操作时间戳 (UTC 时间) | 协议类型 (SOAP/REST/HTTP) | 记录类型 (请求/响应) |
接口名称 | 操作名称 | 源设备名 | 目标设备名 | 消息唯一标识 (traceId) | 本机 IP 地址 | 主叫号码 |
```

被叫号码

图 4-3 展示了一个示例接口日志文件。

```
2016-03-30 08:49:36.571|SOAP|request|SMS|sendSms|OSG|ESB|00200003373|192.168.1.102|13612345678|136712345670
2016-03-30 08:49:36.572|SOAP|response|SMS|sendSmsRsp|ESB|OSG|00200003373|192.168.1.102|13612345678|136712345670
2016-03-30 08:49:36.573|SOAP|request|SMS|sendSms|ESB|NIG|00210003374|192.168.1.102|13612345678|136712345670
2016-03-30 08:49:36.636|SOAP|response|SMS|sendSmsRsp|NIG|ESB|00210003377|192.168.1.102|13612345678|136712345670
2016-03-30 08:49:36.572|SOAP|request|Chg|getPrice|OSG|ESB|00200003350|192.168.1.102|13612345678|136712345670
2016-03-30 08:49:36.574|SOAP|request|Chg|getPrice|ESB|BSS|00210003351|192.168.1.102|13612345678|136712345670
2016-03-30 08:49:37.323|SOAP|response|Chg|getPriceRsp|BSS|ESB|00210003354|192.168.1.102|13612345678|136712345670
2016-03-30 08:49:37.325|SOAP|response|Chg|getLocationRsp|ESB|OSG|00200003352|192.168.1.102|13612345678|136712345670
2016-03-30 08:49:26.575|REST|request|Location|getLocation|OSG|ESB|00200003277|192.168.1.102|13612345678|136712345670
2016-03-30 08:49:26.587|REST|request|Location|getLocation|ESB|NIG|00210003278|192.168.1.102|13612345678|136712345670
```

图 4-3 示例接口日志文件

响应延时统计的算法如下：

以一定的时间（如 10s）为周期，然后将请求时间落在该周期内的指定请求的响应延时累加起来。在输出统计结果的时候，我们只需要逐个地将各个周期的响应延时累加值除以周期长度就可以得到该周期内的平均响应延时。对于单条请求的响应延时我们可以采用这种方法计算：在读取到一个表示请求的日志记录时记录下相应请求的消息唯一标识（traceId）、请求时间戳，然后读取到一条表示响应的日志记录时，根据指定的消息唯一标识差值（traceIdDiff，如 3）计算出与该响应记录对应的请求记录的消息唯一标识，凭此消息唯一标识找到之前存储的请求时间戳，通过计算该响应记录的时间戳与相应的请求时间戳之差就可以得到单条请求的响应延时。

一个接口日志文件最多可以包含 1 万条记录，而我们可能需要从指定的上百个这样的文件进行统计，即统计程序的输入规模（记录条数）可能达到几百万甚至上千万。为了提高统计的效率，这个问题乍一看似乎可以使用 4.3 节我们提到的基于数据的分割。例如，假设指定的接口日志文件个数为 400 个，那么我们可以考虑指定 4 个线程，其中每个线程负责对指定日志文件中的 100 个文件进行统计，即这些工作者线程各自逐条读取 100 个文件中的记录，再根据记录中的数据按照上述算法进行统计。但是这样做也存在以下几个问题。

- **问题 1** 增加程序的复杂性：由于代表一对请求和响应的两条日志记录可能被分别存储在两个日志文件中，因此多个工作者线程并发地读取日志文件的时候可能出现代表响应的日志记录先被读取，而代表相应请求的日志记录后被读取。这样一来实现上述算法就有些困难。
- **问题 2** 可能导致 I/O 资源争用增加而减低 I/O 效率：机械式硬盘在顺序读取文件（读取完一个文件接着再读取另外一个文件）的时候效率会比较高，而多个工作者

线程并发地读取多个文件可能反而会降低文件读取的效率。

- **问题 3** 可能导致处理器时间的浪费：一个工作者线程在等待磁盘返回数据的期间，该线程是处于暂停（WAITING）状态的，它无法执行其他计算，从而导致处理器时间的浪费。

由此可见，在该案例中直接使用基于数据的分割是不太合适的。因此，我们不妨先考虑一下单线程的实现方式。为了便于评估其他的实现方案（比如其他方法实现的多线程版），我们对这个统计程序的算法步骤进行了抽象，如清单 4-5 所示。

清单 4-5 对统计程序算法步骤的抽象

```
/**
 * 对统计程序的算法步骤进行抽象
 *
 * @author Viscent Huang
 */
public abstract class AbstractStatTask implements Runnable {
    private static final String TIME_STAMP_FORMAT = "yyyy-MM-dd HH:mm:ss.SSS";
    private final Calendar calendar;
    // 此处是单线程访问，故其使用是线程安全的
    private final SimpleDateFormat sdf;
    // 采样周期，单位：s
    private final int sampleInterval;
    // 统计处理逻辑类
    protected final StatProcessor recordProcessor;

    public AbstractStatTask(int sampleInterval, int traceIdDiff,
        String expectedOperationName, String expectedExternalDeviceList) {
        this(sampleInterval, new RecordProcessor(sampleInterval,
            traceIdDiff,
            expectedOperationName, expectedExternalDeviceList));
    }

    public AbstractStatTask(int sampleInterval,
        StatProcessor recordProcessor) {
        SimpleTimeZone stz = new SimpleTimeZone(0, "UTC");
        this.sdf = new SimpleDateFormat(TIME_STAMP_FORMAT);
        sdf.setTimeZone(stz);
        this.calendar = Calendar.getInstance(stz);
        this.sampleInterval = sampleInterval;
        this.recordProcessor = recordProcessor;
    }

    /**
     * 留给子类用于实现统计操作的抽象方法
     */
}
```



```

    */
    protected abstract void doCalculate() throws IOException,
        InterruptedException;

    @Override
    public void run() {
        // 执行统计逻辑
        try {
            doCalculate();
        } catch (Exception e) {
            e.printStackTrace();
            return;
        }
        // 获取统计结果
        Map<Long, DelayItem> result = recordProcessor.getResult();
        // 输出统计结果
        report(result);
    }

    protected void report(Map<Long, DelayItem> summaryResult) {
        int sampleCount;
        final PrintStream ps = System.out;
        ps.printf("%s\t\t%s\t\t%s\t\t%s\n",
            "Timestamp", "AvgDelay(ms)", "TPS", "SampleCount");
        for (DelayItem delayStatData : summaryResult.values()) {
            sampleCount = delayStatData.getSampleCount().get();
            ps.printf("%s%8d%8d%8d\n",
                getUTCTimeStamp(delayStatData
                    .getTimeStamp()), delayStatData.getTotalDelay().get()
                    / sampleCount,
                sampleCount
                    / sampleInterval, sampleCount);
        }
    }

    private String getUTCTimeStamp(long timeStamp) {
        calendar.setTimeInMillis(timeStamp);
        String tempTs = sdf.format(calendar.getTime());
        return tempTs;
    }
}

```

这里，我们在 `Runnable` 接口的 `run` 方法中定义了统计程序的算法步骤：执行统计逻辑、获取统计结果和打印统计结果。无论是采用单线程方式还是多线程方式实现这个算法步骤，其中不同的部分只有第 1 步，因此我们使用了抽象方法 `doCalculate` 来表示这个步骤。下面我们以单线程的方式实现这个统计程序，为此我们只需要创建 `AbstractStatTask`

的一个子类 `SimpleStatTask`，并在该子类中实现抽象方法 `doCalculate` 即可，如清单 4-6 所示。

清单 4-6 单线程方式实现的统计程序

```
public class SimpleStatTask extends AbstractStatTask {
    private final InputStream in;

    public SimpleStatTask(InputStream in, int sampleInterval, int traceIdDiff,
        String expectedOperationName, String expectedExternalDeviceList) {
        super(sampleInterval, traceIdDiff, expectedOperationName,
            expectedExternalDeviceList);
        this.in = in;
    }

    @Override
    protected void doCalculate() throws IOException, InterruptedException {
        String strBufferSize = System.getProperty("x.input.buffer");
        int inputBufferSize = null != strBufferSize ? Integer
            .valueOf(strBufferSize) : 8192 * 4;
        final BufferedReader logFileReader = new BufferedReader(
            new InputStreamReader(in), inputBufferSize);
        String record;
        try {
            while ((record = logFileReader.readLine()) != null) {
                // 实例变量 recordProcessor 是在 AbstractStatTask 中定义的
                recordProcessor.process(record);
            }
        } finally {
            Tools.silentClose(logFileReader);
        }
    }
}
```

`SimpleStatTask` 的 `doCalculate` 方法每读取一条日志记录便调用 `recordProcessor` (`RecordProcessor` 类的实例，源码见本书配套下载资源) 的 `process` 方法进行统计处理。显然，`doCalculate` 方法所执行的任务是一个混合型任务。这个任务的执行线程（即 `SimpleStatTask.run()` 的执行线程）在等待磁盘返回数据期间什么也做不了，从而导致处理器时间的浪费。为了提高并发性以提高统计效率，我们可以考虑将这个任务分解为 CPU 密集型和 I/O 密集型两种子任务，并采用专门的工作者线程分别负责这两种子任务的执行。为此，我们只需要再定义一个 `AbstractStatTask` 的子类 `MultithreadedStatTask`，如清单 4-7 所示。

清单 4-7 基于任务分割方式实现的统计程序

```
public class MultithreadedStatTask extends AbstractStatTask {
```

```

// 日志文件输入缓冲大小
protected final int inputBufferSize;
// 日志记录集大小
protected final int batchSize;
// 日志文件输入流
protected final InputStream in;

/* 实例初始化块 */
{
    String strBufferSize = System.getProperty("x.input.buffer");
    inputBufferSize = null != strBufferSize ? Integer.valueOf(strBufferSize)
        : 8192;
    String strBatchSize = System.getProperty("x.batch.size");
    batchSize = null != strBatchSize ? Integer.valueOf(strBatchSize) : 2000;
}

public MultithreadedStatTask(int sampleInterval,
    StatProcessor recordProcessor) {
    super(sampleInterval, recordProcessor);
    this.in = null;
}

public MultithreadedStatTask(InputStream in, int sampleInterval,
    int traceIdDiff,
    String expectedOperationName, String expectedExternalDeviceList) {
    super(sampleInterval, traceIdDiff, expectedOperationName,
        expectedExternalDeviceList);
    this.in = in;
}

@Override
protected void doCalculate() throws IOException, InterruptedException {
    final AbstractLogReader logReaderThread = createLogReader();
    // 启动工作者线程
    logReaderThread.start();
    RecordSet recordSet;
    String record;
    for (;;) {
        recordSet = logReaderThread.nextBatch();
        if (null == recordSet) {
            break;
        }
        while (null != (record = recordSet.nextRecord())) {
            // 实例变量 recordProcessor 是在 AbstractStatTask 中定义的
            recordProcessor.process(record);
        }
    }
} // for 循环结束
}

```

```
protected AbstractLogReader createLogReader() {
    AbstractLogReader logReader = new LogReaderThread(in, inputBufferSize,
        batchSize);
    return logReader;
}
}
```

`MultithreadedStatTask.doCalculate()` 创建并启动了工作者线程 `logReaderThread` (`LogReaderThread` 实例, 源码见清单 4-10), 该线程专门负责日志文件的读取并将其读取到的一批日志记录填充到指定的日志记录集 (`RecordSet` 类的实例, 代表一批日志记录, 源码见清单 4-8) 中。因此, 我们称 `AbstractLogReader` 子类的实例 (`logReaderThread`) 为日志文件读取线程。`MultithreadedStatTask.doCalculate()` 需要读取一批日志记录的时候便调用 `logReaderThread.nextBatch()` 来获取一个已填充完毕的日志记录集, 然后遍历该日志记录集并调用 `RecordProcessor` 的 `process` 方法对遍历到的日志记录进行统计处理。因此, 我们称 `MultithreadedStatTask.doCalculate()` 的执行线程为统计处理线程。由于 `MultithreadedStatTask.doCalculate()` 是由其父类 (`AbstractStatTask`) 的 `run` 方法调用的, 而 `AbstractStatTask.run()` 是由 `main` 线程执行的, 因此 `main` 线程就是统计处理线程。这里, 我们使用唯一的一个线程 (统计处理线程) 负责对日志记录进行统计逻辑处理, 同时又采用另外一个工作者线程 (日志文件读取线程) 负责对日志文件进行读取 (参见清单 4-9)。因此我们自然地绕过了上述问题 2 和问题 1。另外, 由于统计处理线程和日志文件读取线程是并发执行的, 因此我们也绕过了上述问题 3。

清单 4-8 日志记录集 `RecordSet` 源码

```
/**
 * 日志记录集。 包含若干条日志记录
 *
 * @author Viscent Huang
 */
public class RecordSet {
    public final int capacity;
    final String[] records;
    int readIndex = 0;
    int writeIndex = 0;

    public RecordSet(int capacity) {
        this.capacity = capacity;
        records = new String[capacity];
    }

    public String nextRecord() {
        String record = null;
    }
}
```

```
        if (readIndex < writeIndex) {
            record = records[readIndex++];
        }
        return record;
    }

    public boolean putRecord(String line) {
        if (writeIndex == capacity) {
            return true;
        }
        records[writeIndex++] = line;
        return false;
    }

    public void reset() {
        readIndex = 0;
        writeIndex = 0;
        for (int i = 0, len = records.length; i < len; i++) {
            records[i] = null;
        }
    }

    public boolean isEmpty() {
        return 0 == writeIndex;
    }
}
```

清单 4-9 日志文件读取线程 AbstractLogReader 源码

```
/**
 * 日志文件读取线程
 *
 * @author Viscent Huang
 */
public abstract class AbstractLogReader extends Thread {
    protected final BufferedReader logFileReader;
    // 表示日志文件是否读取结束
    protected volatile boolean isEOF = false;
    protected final int batchSize;

    public AbstractLogReader(InputStream in, int inputBufferSize, int batchSize) {
        logFileReader = new BufferedReader(new InputStreamReader(in),
            inputBufferSize);
        this.batchSize = batchSize;
    }

    protected RecordSet getNextToFill() {
```

```

    return new RecordSet(batchSize);
}

/* 留给子类实现的抽象方法 */
// 获取下一个记录集
protected abstract RecordSet nextBatch()
    throws InterruptedException;

// 发布指定的记录集
protected abstract void publish(RecordSet recordBatch)
    throws InterruptedException;

@Override
public void run() {
    RecordSet recordSet;
    boolean eof = false;
    try {
        while (true) {
            recordSet = getNextToFill();
            recordSet.reset();
            eof = doFill(recordSet);
            publish(recordSet);
            if (eof) {
                if (!recordSet.isEmpty()) {
                    publish(new RecordSet(1));
                }
                isEOF = eof;
                break;
            }
        }
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        Tools.silentClose(logFileReader);
    }
}

protected boolean doFill(final RecordSet recordSet) throws IOException {
    final int capacity = recordSet.capacity();
    String record;
    for (int i = 0; i < capacity; i++) {
        record = logFileReader.readLine();
        if (null == record) {
            return true;
        }
        // 将读取到的日志记录存入指定的记录集
        recordSet.putRecord(record);
    }
}

```

```

    return false;
}
}

```

综上所述，本案例实际上是将程序算法中的读取日志文件和对日志记录进行统计处理这两个步骤由原来的单线程程序的串行处理改为并发处理，即实现了并发化。

下面我们比较一下本案例的单线程版程序与多线程版程序统计效率的差异。

使用如下命令运行本案例的单线程版统计程序：

```

perf stat -e cs,cpu-clock,task-clock \
java -Xms96m -Xmx128m -XX:NewSize=64m -XX:SurvivorRatio=32 -Dx.std.in="/tmp/in.d
at" -Dx.stat.task=io.github.viscent.mtia.ch4.case02.SimpleStatTask -Dx.input.buf
fer=8192 io.github.viscent.mtia.ch4.case02.CaseRunner4_2

```

可以看到类似如下的输出（省略部分输出）：

```

7,059 cs # 0.000 M/sec
17188.727772 cpu-clock
17188.713640 task-clock # 1.001 CPUs utilized

17.174350663 seconds time elapsed

```

使用如下命令运行本案例的多线程版统计程序：

```

perf stat -e cs,cpu-clock,task-clock \
java -Xms96m -Xmx128m -XX:NewSize=64m -XX:SurvivorRatio=32 -Dx.std.in="/tmp/in.d
at" -Dx.stat.task=io.github.viscent.mtia.ch4.case02.MultithreadedStatTask -Dx.in
put.buffer=8192 -Dx.batch.size=2000 io.github.viscent.mtia.ch4.case02.CaseRunner
4_2

```

可以看到类似如下的输出（省略部分输出）：

```

9,197 cs # 0.001 M/sec
18154.625696 cpu-clock
18154.909540 task-clock # 1.860 CPUs utilized

9.760901177 seconds time elapsed

```

可见，多线程版的统计程序比单线程版的统计程序要快（当然，这是有代价的，下文会讲到这一点），其提速约为 $1.76 (=17174/9760)^3$ 。这个值已经接近该程序的最大理论提

3 这个对比实验的环境为：输入记录总数为 6 480 000 条，Linux 32 位系统，1 颗双核 CPU，机械式硬盘。另外，由于这个实验涉及 I/O，因此在对比时需要注意先让操作系统的 I/O 操作“预热”，即先让单线程程序和多线程程序各自都运行一遍，然后再分别运行单线程程序和多线程程序才能够对比二者

速(=2): 假设该统计程序的单线程版中读取日志文件总耗时为 P_1 , 对日志记录进行统计处理的总耗时为 P_2 , 那么由于该程序中可以并发化的部分也只有读取日志文件和对日志记录进行统计处理这两个部分, 因此将该程序并发化所能达到的最大理论提速(单线程版程序耗时与多线程版程序耗时之比)如下⁴:

$$S_{\max} = \frac{P_1 + P_2}{\max(P_1, P_2)} \leq 2 \quad (4-1)$$

其中, $\max(P_1, P_2)$ 表示 P_1 、 P_2 的最大值。

4.4.2 实战案例的启发

从上述案例中可以看出, 基于任务的分割这种并发化策略是从程序的处理逻辑角度入手, 将原始任务处理逻辑分解为若干子任务, 并创建专门的工作者线程来执行这些子任务。基于任务的分割结果是产生多个相互协作的异质工作者线程, 即任务处理逻辑各异的线程。例如, 在上述案例中日志文件读取线程(见清单 4-9)负责日志文件记录的读取, 而统计处理线程(main 线程)则负责对读取到内存中的日志记录集进行统计, 这两种(个)线程的任务处理逻辑不同, 它们相互协作来完成原始任务的处理。

另外, 需要注意以下几点。

- 基于任务的分割同样可能导致程序的复杂性增加。日志统计处理线程与日志读取线程之间的协作本身就增加了程序的复杂性。例如, 在上述案例中我们需要借助一个线程安全队列 `java.util.concurrent.ArrayBlockingQueue` (第 5 章会介绍该类) 来实现日志统计处理线程与日志读取线程之间的数据(日志记录集)交互, 如清单 4-10 所示。

清单 4-10 日志文件读取线程实现类 `LogReaderThread` 源码

```
/**
 * 日志读取线程实现类
 *
 * @author Viscent Huang
 */
public class LogReaderThread extends AbstractLogReader {
    // 线程安全的队列
    final BlockingQueue<RecordSet> channel = new ArrayBlockingQueue<RecordSet>(2);
```

的真正性能差异(即排除了操作系统 I/O 操作是否被“预热”而导致的差异)。

4 多线程版的统计程序的耗时取决于 P_1 和 P_2 中的最大值。


```

public LogReaderThread(InputStream in, int inputBufferSize, int batchSize) {
    super(in, inputBufferSize, batchSize);
}

@Override
public RecordSet nextBatch()
    throws InterruptedException {
    RecordSet batch;
    // 从队列中取出一个记录集
    batch = channel.take();
    if (batch.isEmpty()) {
        batch = null;
    }
    return batch;
}

@Override
protected void publish(RecordSet recordBatch) throws InterruptedException {
    // 记录集存入队列
    channel.put(recordBatch);
}
}

```

上述案例中待统计的日志记录可能多达上千万条，因此如果日志统计处理线程是逐条地从日志读取线程读取日志记录，那么这两个线程之间的数据传递（移动）的开销将不容小觑。所以，我们使用 `RecordSet` 类作为日志统计处理线程和日志文件读取线程之间数据传递的容器。该容器使得一批日志记录（例如 2000 条）成为日志统计处理线程和日志文件读取线程之间的数据传递单位，从而减少了数据传递的开销，而这同时也在一定程度上增加了程序的复杂性。

- 多线程程序可能增加额外的处理器时间消耗。由于多线程版的统计程序比相应的单线程版程序增加了工作者线程的创建与启动、日志记录集的填充、日志文件读取线程和日志记录统计处理线程之间的数据传递以及额外的上下文切换等开销，多线程版的统计程序运行时的处理器时间消耗要比相应的单线程程序多了不少⁵。
- 多线程程序未必比相应的单线程程序快。从上文可知，多线程版的统计程序的确是比相应的单线程版程序要快一些，但实际上也可能不是快很多。这是因为影响程序性能的因素是多方面的，而多线程与单线程的差别只是其中一方面而已！例如，上述案例中程序所使用的两个重要参数是日志文件读取线程所使用的文件输入缓冲区大小（通过自定义的虚拟机系统属性“`x.input.buffer`”指定）和日志记

5 Java 平台本身就是一个多线程的平台，因此我们所说的单线程程序实际上是指应用程序自身没有创建其线程。

录缓冲区的容量（通过自定义的虚拟机系统属性“x.batch.size”指定）。这两个参数值对多线程版的统计程序的性能有着关键性的影响——这两个参数设置得不合理（比如日志记录缓冲区的容量过小）可能使得多线程版的统计程序比单线程版的还慢。另外，还有一些一般性因素也会影响 Java 程序的性能，包括垃圾回收（它可能导致上下文切换）、JIT 动态编译（动态编译也有自身的开销）等。

- 考虑从单线程程序向多线程程序“进化”。考虑到多线程程序的相对复杂性以及多线程程序未必比单线程程序要快，使用多线程编程的一个好的方式是从单线程程序开始，只有在单线程程序算法本身没有重大性能瓶颈但仍然无法满足要求的情况下我们才考虑使用多线程。当然，这个过程需要注意重复建设的问题。这一点可以通过在代码中采用一定程度的抽象来避免或者减少这方面的成本。例如，在上述案例中，我们用 `AbstractStatTask` 这个抽象类（参见清单 4-5）对统计程序的算法步骤进行了抽象，这使得从单线程程序向多线程程序“进化”时无须修改现有代码，而只需要新建一个 `AbstractStatTask` 的子类，在该子类中实现多线程编程。这种方式不仅减少了编码量，还避免了对现有代码进行重复调试、测试！

提示

使用多线程编程的一个好的方式是从单线程程序开始，只有在单线程程序算法本身没有重大性能瓶颈但仍然无法满足要求的情况下我们才考虑使用多线程。

4.4.3 按处理步骤分割

如果程序对其输入集 $\{d_1, d_2, \dots, d_N\}$ 中的任何一个输入元素 $d_i (1 \leq i \leq N)$ 的处理都包含若干步骤 $\{\text{Step}_1, \text{Step}_2, \dots, \text{Step}_M\}$ ，那么为了提高程序的吞吐率，我们可以考虑为其中的每一个处理步骤都安排一个（或者多个）工作者线程负责相应的处理。这就是按处理步骤分割的基本思想。

按任务的资源消耗属性分割可以被看作按处理步骤分割的一个特例。多线程设计模式中的 Pipeline 模式的核心思想也正是按处理步骤分割的。

在按处理步骤分割实现的多线程程序中，多个工作者线程并发地对程序输入集中的不同输入元素进行处理，这提高了程序的吞吐率。这好比上文提到的饭店招待客人的例子中，多个服务员（工作者线程）同时（并发）服务多个顾客（输入元素）可以提高饭店的接待能力（吞吐率）。类似于按任务的资源消耗属性分割，在按处理步骤分割中工作者线程之间传递数据同样也需要借助线程安全的队列，而这也会增加相应的开销。因此，按处理步骤分割可能导致单个输入元素的处理时间相对变大，即延迟增加。

同样，在按处理步骤分割中我们 also 需要注意工作者线程数的合理设置：工作者线程数

量过多可能会导致过多的上下文切换，这反而降低了程序的吞吐率。因此，保守的设置方法是从仅为每个处理步骤设置一个工作者线程开始，在确实有证据显示有必要增加某个处理步骤的工作者线程数的情况下才增加线程数。

4.5 合理设置线程数

在本章的第一个案例中，工作者线程的数量是通过程序的参数指定的（见本书配套下载资源中的类 `CaseRunner4_1`）。线程数不宜过小，线程数过小可能导致无法充分利用处理器资源；线程数也不宜过大，线程数过大会增加上下文切换以及其他开销。那么，我们如何设置一个合理的线程数呢？在回答这个问题之前，我们先看一下线程数与多线程程序相对于单线程程序的提速（Speedup）之间的关系。

4.5.1 Amdahl's 定律

Amdahl's 定律（Amdahl's Law）描述了线程数与多线程程序相对于单线程程序的提速之间的关系。在一个处理器上一个时刻只能够运行一个线程的情况下，处理器的数量就等同于并行线程的数量。设处理器的数量为 N ，程序中必须串行（即无法并发化）的部分耗时占程序全部耗时的比率为 P ，那么将这样一个程序改为多线程程序，我们能够获得的理论上的最大提速 S_{\max} 与 N 、 P 之间的关系就是 Amdahl's 定律内容，如下所示。

$$S_{\max} = \frac{1}{P + \frac{1-P}{N}} \quad (4-2)$$

了解该公式的推导过程有助于我们更好地理解多线程编程的本质。我们知道，一个程序的算法中有些部分是可以并行化的，而有些部分则只能是串行的。设 P 为这个程序的串行部分的耗时比率， $T(1)$ 为该程序的单线程版运行总耗时， $T(N)$ 为该程序的多线程版运行总耗时，那么将该程序由单线程改为多线程所得到的提速 S_{\max} 可以表示为：

$$S_{\max} = \frac{T(1)}{T(N)} \quad (4-3)$$

为方便起见，设 $T(1)$ 为 1，则该程序中的串行部分耗时为 P ，可并行部分耗时为 $1-P$ 。将这个程序改为多线程程序的时候，该程序的可并行部分耗时会被 N 个并行线程平均分摊，因此该程序的多线程版的并行部分总耗时为 $(1-P)/N$ （串行部分仍然是 P ！）。由此，我们可以得出 $T(N) = P + (1-P)/N$ 。将 $T(N)$ 及 $T(1)=1$ 代入式（4-2）即可得到 Amdahl's 定律的

公式表示。

从上述推导过程可以看出，多线程程序的提速主要来自多个线程对程序中可并行化部分的耗时均摊。

由 Amdahl's 定律的公式可知：

$$\lim_{N \rightarrow \infty} S_{\max} = \lim_{N \rightarrow \infty} \frac{1}{P + \frac{1-P}{N}} = \frac{1}{P} \quad (4-4)$$

即当 N 趋向于无穷大的时候， S_{\max} 趋向于 $1/P$ 。由此可见，最终决定多线程程序提速的因素是整个计算中串行部分的耗时比率 P ，而不是线程数 N ！ P 的值越大，即程序中不可并行化的部分所占比率越大，那么提速越小。因此，为使多线程程序能够获得较大的提速，我们应该从算法入手，减少程序中必须串行的部分，而不是仅寄希望于增加线程数（或者处理器的数目）！

4.5.2 线程数设置的原则

线程数设置得过少可能导致无法充分利用处理器资源；而线程数设置得过大则又可能导致过多的上下文切换，从而反倒降低了系统的性能。然而，设置一个“既不过小，也不过大”的绝对理想的线程数实际上是不可能的。这是因为设置一个绝对理想的线程数所需的信息对我们来说总是不充分的。这就好比“石头剪子布”这个游戏中，由于我们无法准确地预料对方下一次会出什么手势，因此我们决定出的下一个手势也并不总是最佳的一样。因此，设置一个合理的线程数实际上就是避免随意设置线程，即在设置线程数时尽可能地考虑一些可以实际操作的因素。这些因素包括系统的资源状况（处理器数目、内存容量等）、线程所执行的任务的特性（CPU 密集型任务、I/O 密集型任务）、资源使用情况规划（CPU 使用率上限）以及程序运行过程中使用到的其他稀缺资源情况（如数据库连接、文件句柄数）等。

由于线程运行的硬件基础是处理器，因此设置线程数首先并且必须要考虑的一个因素就是系统的处理器数目。但是，由于一个系统的处理器资源可能是由上百个进程共享的，因此，要为一个应用程序设置合理的线程数，从理论上说我们需要考虑其他所有进程内部的线程数设置情况。显然，这是不可能实现的，因为我们无法事先知道一个环境（尤其是生产环境）的进程数量。退一步来讲，即便是在一个应用程序（进程），例如一个 Java Web 应用程序中，从一个模块出发来考虑该模块线程数的合理值，要将该程序的其他模块的线程数设置情况考虑进来也不是一件容易的事情。因此，所谓合理设置线程数仅仅是指避免

随意设置而已，我们无法达到一个纯粹的理想值。

设 N_{cpu} 表示一个系统的处理器数目， N_{cpu} 的具体值可以通过如下 Java 代码获取：

```
int nCPU = Runtime.getRuntime().availableProcessors();
```

线程数的合理值可以根据以下规则设置。

- 对于 CPU 密集型线程，考虑到这类线程执行任务时消耗的主要是处理器资源，我们可以将这类线程的线程数设置为 N_{cpu} 个。因为 CPU 密集型线程也可能由于某些原因（比如缺页中断/Page Fault）而被切出，此时为了避免处理器资源浪费，我们也可以为这类线程设置一个额外的线程，即将线程数设置为 $N_{\text{cpu}}+1$ 。
- 对于 I/O 密集型线程，考虑到 I/O 操作可能导致上下文切换，为这样的线程设置过多的线程数会导致过多的额外系统开销。因此如果一个这样的工作者线程就足以满足我们的要求，那么就不要设置更多的线程数。例如，在本章的第 2 个实战案例中我们仅使用一个工作者线程去负责所有日志文件的读取。如果一个工作者线程仍然不够用，那么我们可以考虑将这类线程的数量设置为 $2 \times N_{\text{cpu}}$ 。这是因为 I/O 密集型线程在等待 I/O 操作返回结果时是不占用处理器资源的，因此我们可以为每个处理器安排一个额外的线程以提高处理器资源的利用率。

提示

对于 CPU 密集型线程，线程数通常可以设置为 $N_{\text{cpu}}+1$ ；对于 I/O 密集型线程，优先考虑将线程数设置为 1，仅在一个线程不够用的情况下将线程数向 $2 \times N_{\text{cpu}}$ 靠近。

商用软件往往会规定某个软件在其运行过程中对处理器的使用率不能超过某个阈值（如 75%）。因此，如果要进一步“精确”地设置线程数，我们可能需要考虑目标处理器使用率，即我们期望软件运行过程中会保持多少平均 CPU 使用率。另外，如果任务本身是混合型而又不太好将其拆分成 CPU 密集型和 I/O 密集型的子任务的话，也可以考虑不拆分。此时，我们可以参考式（4-5）来设置线程数：

$$N_{\text{threads}} = N_{\text{cpu}} \times U_{\text{cpu}} \times \left(1 + \frac{\text{WT}}{\text{ST}} \right) \quad (4-5)$$

其中， N_{threads} 为线程数的合理大小， N_{cpu} 为 CPU 数目， U_{cpu} 为目标 CPU 使用率（ $0 < U_{\text{cpu}} \leq 1$ ），WT（Wait Time）为程序花费在等待（例如等待 I/O 操作结果）上的时长，ST（Service Time）为程序实际占用处理器执行计算的时长。在实践中，我们可以使用 jvisualvm 提供的监控数据计算出 WT/ST 的值。

下面通过一个 Demo 介绍 WT/ST 的计算。使用如下命令运行如清单 4-11 所示的 Demo：

```
java -Dx.prepause=40000 -Dx.postpause=10000 io.github.viscent.mtia.util.AppWrapp
er io.github.viscent.mtia.ch4.WTSTMeasureDemo
```

并启动 jvisualvm 对上述程序的 CPU 使用情况进行采样。

清单 4-11 实际测量 WT/ST 值 Demo

```
public class WTSTMeasureDemo implements Runnable {
    final long waitTime;

    public WTSTMeasureDemo(long waitTime) {
        this.waitTime = waitTime;
    }

    public static void main(String[] args) throws Exception {
        main0(args);
    }

    public static void main0(String[] args) throws Exception {
        final int argc = args.length;
        int nThreads = argc > 0 ? Integer.valueOf(args[0]) : 1;
        long waitTime = argc >= 1 ? Long.valueOf(args[0]) : 4000L;
        WTSTMeasureDemo demo = new WTSTMeasureDemo(waitTime);
        Thread[] threads = new Thread[nThreads];
        for (int i = 0; i < nThreads; i++) {
            threads[i] = new Thread(demo);
        }
        long s = System.currentTimeMillis();
        Tools.startAndWaitTerminated(threads);
        long duration = System.currentTimeMillis() - s;
        long serviceTime = duration - waitTime;
        System.out.printf(
            "WT/ST: %-4.2f, waitTime: %dms, serviceTime: %dms, duration: %4.2fs%n",
            waitTime * 1.0f / serviceTime,
            waitTime, serviceTime,
            duration * 1.0f / 1000);
    }

    @Override
    public void run() {
        try {
            // 模拟 I/O 操作
            Thread.sleep(waitTime);

            // 模拟实际执行计算
            String result = null;
            for (int i = 0; i < 400000; i++) {
                result = DESEncryption.encryptAsString(
```

```

        "it is a cpu-intensive task" + i,
        "12345678");
    }
    System.out.printf("result:%s\n", result);
} catch (Exception e) {
    e.printStackTrace();
}
}
}
}

```

该程序的输出类似如下：

```

Arguments:[io.github.viscent.mtia.ch4.WTSTMeasureDemo]
Started at:2016-11-15 15:29:31.102
result:JhtqYJVVV+urCWHcAY/jmnX2dVJo9Acqhee72tZHFq7+uVm31GQvyw==
WT/ST: 1.21, waitTime: 4000ms, serviceTime: 3297ms, duration: 7.30s
Finished at:2016-11-15 15:29:38.314
Time consumed:7301ms

```

可见，WT/ST 的值为 1.21。当然，利用 jvisualvm 提供的监控数据来计算该值会比较方便。该程序的 jvisualvm 采样结果如图 4-4 所示。

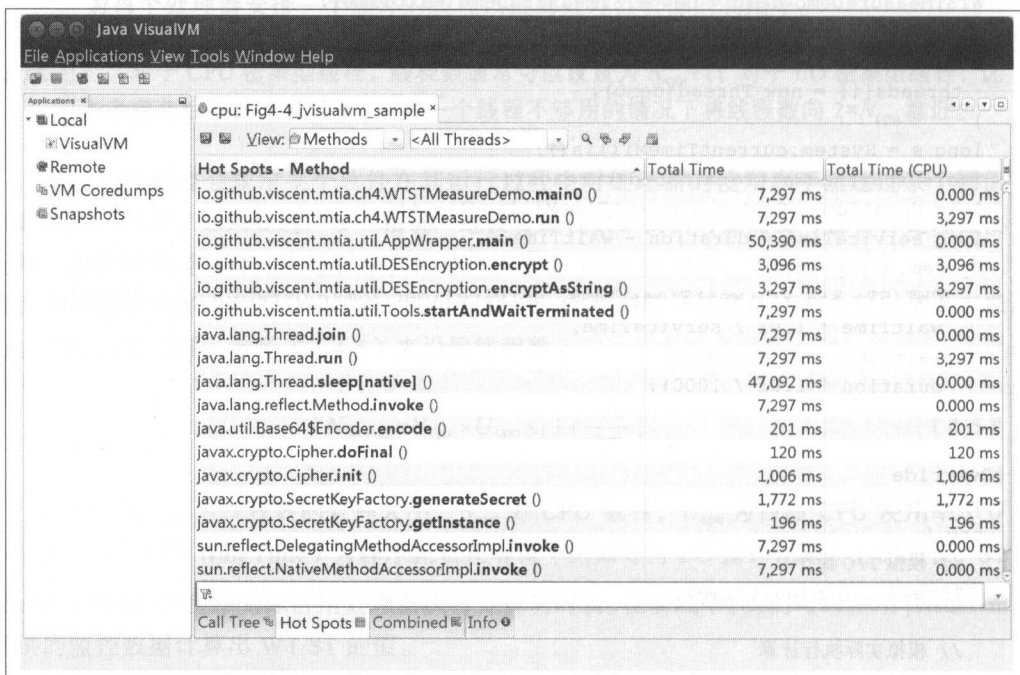


图 4-4 jvisualvm 采样结果

从图 4-4 可见，WTSTMeasureDemo.run()执行总耗时（Total Time）7297 毫秒，而其消耗的处理器时间（Total Time(CPU)）为 3297 毫秒。由此可知 WT/ST 为： $(7297-3297)/3297 \approx 1.21$ 。这个值与我们通过代码计算出来的值相吻合。

另外，我们还可能要考虑线程执行过程中使用到的其他稀缺资源的情况，如数据库连接、文件句柄数、网络连接等。比如，与数据库打交道的任务在执行过程中需要获取数据库连接。虽然数据库连接多数是从数据库连接池中取得的，而不是每次执行这种任务都新建一个数据库连接；但是从数据库服务器的角度来看，一个数据库服务器能够对外提供的数据库连接始终是有限的。在应用程序采用集群（Cluster）部署的情况下，可能有多台主机共同访问同一个数据库服务器。因此，对该集群环境中的一台主机而言，其能够使用的数据库连接资源就显得更加有限。所以，此时线程数的合理大小实际上还要受可用的数据库连接数的限制。

从实践上看，通常我们可以采用配置的方式（如配置文件）或者通过代码自动计算的方式来设置线程数，而不是将线程数硬编码（Hard-coded）在代码之中。例如，本章第一个实战案例就是采用在程序参数中指定工作者线程数这种方式来设置线程数的。

4.6 本章小结

本章介绍了利用多线程实现并发计算的基本方法以及多线程编程实践中的注意事项及应对措施。本章知识结构如图 4-5 所示。

挖掘出程序中的可并发点是实现多线程编程的目标——并发计算的前提。

实现并发化的策略包括基于数据的分割策略和基于任务的分割策略。前者从程序处理的数据角度入手，将原始输入分解为若干规模更小的子输入，并将这些子输入指派给专门的工作者线程处理。其结果是产生若干同质的工作者线程。后者从程序的处理逻辑角度入手，将原始任务处理逻辑依照任务的资源消耗属性或者处理步骤分解为若干子任务，并创建专门的工作者线程来执行这些子任务。其结果是产生多个相互协作的异质工作者线程。

多线程编程实践中需要注意以下几个问题。

- 考虑到多线程程序往往比相应的单线程程序要复杂，且未必比相应的单线程程序快，因此多线程编程的一个实施策略是考虑从单线程程序向多线程程序“进化”，而不是直接迈向“多线程”。

- 线程数的合理设置。设置线程数的基本原则就是避免随意设置、使线程数可配置或者可以动态计算得来。设置合理的线程数需要考虑系统的资源状况（处理器数目、内存大小等）、线程所执行的任务的特性（CPU 密集型任务、I/O 密集型任务）、资源使用情况规划（CPU 使用率上限）以及程序运行过程中使用到的其他稀缺资源情况（如数据库连接、文件句柄数）等因素。
- 多线程程序往往比相应的单线程程序产生更多的开销，且需要注意工作者线程的异常处理以及原始任务规模未知问题的应对。

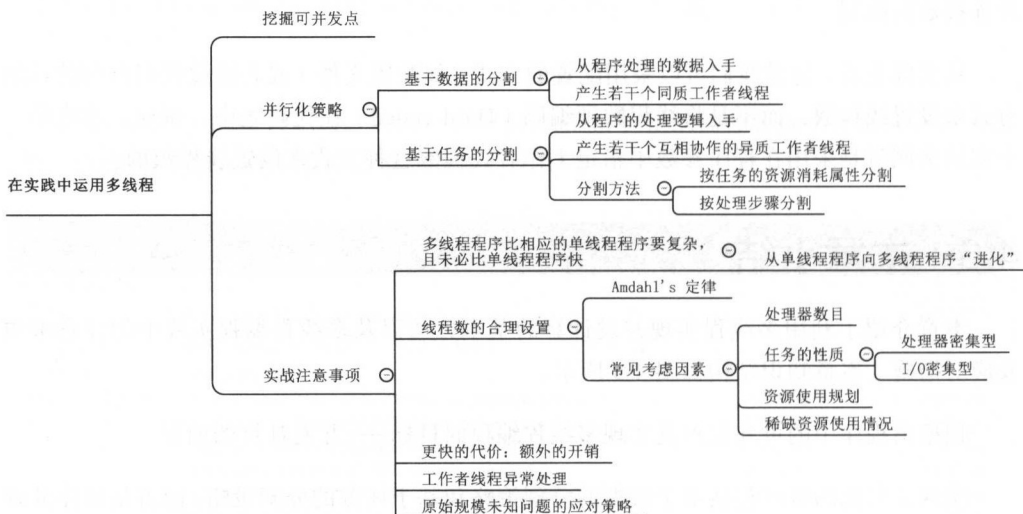


图 4-5 本章知识结构图

线程间协作

你不是一个人在战斗!

——黄健翔

面向对象的世界中类不是孤立的，一个类往往需要借助其他类才能完成一个计算。同样，多线程世界中的线程并不是孤立的，一个线程往往需要其他线程的协作才能够完成其待执行的任务。本章将介绍线程间的常见协作形式以及 Java 语言对这些协作所提供的支持。

5.1 等待与通知：wait/notify

在单线程编程中，程序要执行的操作（目标动作）如果需要满足一定的条件（保护条件）才能执行，那么我们可以将该操作放在一个 if 语句体中，这使得目标动作只有在保护条件得以满足的时候才会被执行。在多线程编程中处理这种情形我们有另外一个选择——保护条件未满足可能只是暂时的，稍后其他线程可能更新了保护条件涉及的共享变量而使得其成立，因此我们可以将当前线程暂停，直到其所需的保护条件得以成立时再将其唤醒，如下伪代码所示：

```
// 原子操作
atomic{
    while(保护条件不成立){
        暂停当前线程;
    }
    // 执行目标动作
    doAction();
}
```

显然，上述操作必须具有原子性。这里，一个线程因其执行目标动作所需的保护条件未满足而被暂停的过程就被称为等待（Wait）。一个线程更新了系统的状态，使得其他线程所需的保护条件得以满足的时候唤醒那些被暂停的线程的过程就被称为通知（Notify）。

5.1.1 wait/notify 的作用与用法

在 Java 平台中，Object.wait()/Object.wait(long)以及 Object.notify()/Object.notifyAll() 可用于实现等待和通知：Object.wait()的作用是使其执行线程被暂停（其生命周期状态变更为 WAITING），该方法可用来实现等待；Object.notify()的作用是唤醒一个被暂停的线程，调用该方法可实现通知。相应地，Object.wait()的执行线程就被称为等待线程；Object.notify()的执行线程就被称为通知线程。由于 Object 类是 Java 中任何对象的父类，因此使用 Java 中的任何对象都能够实现等待与通知。

使用 Object.wait()实现等待，其代码模板如下伪代码所示：

```
// 在调用 wait 方法前获得相应对象的内部锁
synchronized(someObject){
    while(保护条件不成立){
        // 调用 Object.wait() 暂停当前线程
        someObject.wait();
    }

    // 代码执行到这里说明保护条件已经满足
    // 执行目标动作
    doAction();
}
```

其中，保护条件是一个包含共享变量的布尔表达式。当这些共享变量被其他线程（通知线程）更新之后使相应的保护条件得以成立时，这些线程会通知等待线程。由于一个线程只有在持有一个对象的内部锁的情况下才能够调用该对象的 wait 方法，因此 Object.wait() 调用总是放在相应对象所引导的临界区之中。包含上述模板代码的方法被称为受保护方法（Guarded Method）。受保护方法包括 3 个要素：保护条件、暂停当前线程和目标动作。

设 someObject 为 Java 中任意一个类的实例，因执行 someObject.wait()而被暂停的线程就称为对象 someObject 上的等待线程。由于同一个对象的同一个方法（someObject.wait()）可以被多个线程执行，因此一个对象可能存在多个等待线程。someObject 上的等待线程可以通过其他线程执行 someObject.notify()来唤醒。someObject.wait()会以原子操作的方式使其执行线程（当前线程）暂停并使该线程释放其持有的 someObject 对应的内部锁。当前线程被暂停的时候其对 someObject.wait()的调用并未返回。其他线程在该线程所需的保护条件成立的时候执行相应的 notify 方法，即 someObject.notify()可以唤醒 someObject 上的一个（任意的）等待线程。被唤醒的等待线程在其占用处理器继续运行的时候，需要再次申请 someObject 对应的内部锁。被唤醒的线程在其再次持有 someObject 对应的内部锁的情况下继续执行 someObject.wait()中剩余的指令，直到 wait 方法返回。

等待线程只在保护条件不成立的情况下才执行 `Object.wait()` 进行等待，即在执行 `Object.wait()` 前我们需要判断保护条件是否成立（当然，此时保护条件也是有可能成立的）。另外，等待线程在其被唤醒、继续运行到其再次持有相应对象的内部锁的这段时间内，由于其他线程可能抢先获得相应的内部锁并更新了相关共享变量而导致该线程所需的保护条件又再次不成立¹，因此 `Object.wait()` 调用返回之后我们需要再次判断此时保护条件是否成立。所以，对保护条件的判断以及 `Object.wait()` 调用应该放在循环语句之中，以确保目标动作只有在保护条件成立的情况下才能够执行！

另外，等待线程对保护条件的判断以及目标动作的执行必须是个原子操作，否则可能产生竞态——目标动作被执行前的那一刻其他线程对共享变量的更新又使得保护条件重新不成立。因此，目标动作的执行必须和保护条件的判断以及 `Object.wait()` 调用放在同一个对象所引导的临界区中。

注意

- 等待线程对保护条件的判断、`Object.wait()` 的调用总是应该放在相应对象所引导的临界区中的一个循环语句之中。
- 等待线程对保护条件的判断、`Object.wait()` 的执行以及目标动作的执行必须放在同一个对象（内部锁）所引导的临界区之中。
- `Object.wait()` 暂停当前线程时释放的锁只是与该 `wait` 方法所属对象的内部锁。当前线程所持有的其他内部锁、显式锁并不会因此而被释放。

使用 `Object.notify()` 实现通知，其代码模板如下伪代码所示：

```
synchronized(someObject){
    // 更新等待线程的保护条件涉及的共享变量
    updateSharedState();
    // 唤醒其他线程
    someObject.notify();
}
```

包含上述模板代码的方法被称为通知方法，它包含两个要素：更新共享变量、唤醒其他线程。由于一个线程只有在持有一个对象的内部锁的情况下才能够执行该对象的 `notify` 方法，因此 `Object.notify()` 调用总是放在相应对象内部锁所引导的临界区之中。也正是由于 `Object.notify()` 要求其执行线程必须持有该方法所属对象的内部锁，因此 `Object.wait()` 在暂停其执行线程的同时必须释放相应的内部锁；否则通知线程无法获得相应的内部锁，也就无法执行相应对象的 `notify` 方法来通知等待线程！`Object.notify()` 的执行线程持有的相应对象的内部锁只有在 `Object.notify()` 调用所在的临界区代码执行结束后才会被释放，而

¹ 下文会介绍到其他因素也会导致此时保护条件再次不成立。

`Object.notify()`本身并不会将这个内部锁释放。因此，为了使等待线程在其被唤醒之后能够尽快再次获得相应的内部锁，我们要尽可能地将 `Object.notify()`调用放在靠近临界区结束的地方。等待线程被唤醒之后占用处理器继续运行时，如果有其他线程持有了相应对象的内部锁，那么这个等待线程可能又会再次被暂停，以等待再次获得相应内部锁的机会，而这会导致上下文切换。

调用 `Object.notify()`所唤醒的线程仅是相应对象上的一个任意等待线程，所以这个被唤醒的线程可能不是我们真正想要唤醒的那个线程。因此，有时候我们需要借助 `Object.notify()`的兄弟——`Object.notifyAll()`，它可以唤醒相应对象上的所有等待线程。由于等待线程和通知线程在其实现等待和通知的时候必须是调用同一个对象的 `wait` 方法、`notify` 方法，而这两个方法都要求其执行线程必须持有该方法所属对象的内部锁，因此等待线程和通知线程是同步在同一对象之上的两种线程。

注意 等待线程和通知线程必须调用同一个对象的 `wait` 方法、`notify` 方法来实现等待和通知。调用一个对象的 `notify` 方法所唤醒的线程仅是该对象上的一个任意等待线程。`notify` 方法调用应该尽可能地放在靠近临界区结束的地方。

扩展阅读 `Object.wait()/notify()`的内部实现

我们知道 Java 虚拟机会为每个对象维护一个入口集（Entry Set）用于存储申请该对象内部锁的线程。此外，Java 虚拟机还会为每个对象维护一个被称为等待集（Wait Set）的队列，该队列用于存储该对象上的等待线程。`Object.wait()`将当前线程暂停并释放相应内部锁的同时会将当前线程（的引用）存入该方法所属对象的等待集中。执行一个对象的 `notify` 方法会使该对象的等待集中的一个任意线程被唤醒。被唤醒的线程仍然会停留在相应对象的等待集之中，直到该线程再次持有相应内部锁的时候（此时 `Object.wait()`调用尚未返回）`Object.wait()`会使当前线程从其所在的等待集移除，接着 `Object.wait()`调用就返回了。`Object.wait()/notify()`实现的等待/通知中的几个关键动作，包括将当前线程加入等待集、暂停当前线程、释放锁以及将唤醒后的等待线程从等待集中移除等，都是在 `Object.wait()`中实现的。`Object.wait()`的部分内部实现相当于如下伪代码：

```
public void wait(){
    // 执行线程必须持有当前对象对应的内部锁
    if(!Thread.holdsLock(this)){
        throws new IllegalMonitorStateException();
    }

    if(当前对象不在等待集中){
        // 将当前线程加入当前对象的等待集中
```

```

    addToWaitSet(Thread.currentThread());
}

atomic{// 原子操作开始
    // 释放当前对象的内部锁
    releaseLock(this);
    // 暂停当前线程
    block(Thread.currentThread()); // 语句①
} // 原子操作结束

// 再次申请当前对象的内部锁
acquireLock(this); // 语句②
// 将当前线程从当前对象的等待集中移除
removeFromWaitSet(Thread.currentThread());
return; // 返回
}

```

等待线程在语句①被执行之后就被暂停了。被唤醒的线程在其占用处理器继续运行的时候会继续执行其暂停前调用的 `Object.wait()` 中的其他指令，即从上述代码中的语句②开始继续执行：先再次申请 `Object.wait()` 所属对象的内部锁，接着将当前线程从相应的等待集中移除，然后 `Object.wait()` 调用才返回！

下面看一个实战案例。某分布式系统有个告警功能模块。该模块的主要职责是接收其他模块提交的告警消息，并将这些告警消息通过网络连接上报（发送）到告警服务器上。该模块中的告警代理（`AlarmAgent` 类，如清单 5-1 所示）负责与告警服务器建立网络连接，并对外暴露一个 `sendAlarm` 方法用于将指定的告警消息上报到告警服务器上。`AlarmAgent` 内部会维护两个工作者线程：一个工作者线程负责与告警服务器建立网络连接（Socket 连接），我们称该线程为网络连接线程；另外一个工作者线程负责定时检测告警代理与告警服务器的网络连接状况，我们称该线程为心跳线程。告警模块还专门维护了一个告警发送线程，该工作者线程通过调用 `AlarmAgent.sendAlarm(String)` 将该模块接收到的告警消息上报给告警服务器。

由于告警发送线程执行 `AlarmAgent.sendAlarm(String)` 的时候 `AlarmAgent` 与告警服务器的网络连接可能尚未建立完毕，或者中途由于一些故障（比如告警服务器宕机）连接已经中断，因此该线程需要等待 `AlarmAgent` 与告警服务器的连接完毕或者恢复连接之后才能上报告警消息，否则会导致告警上报失败。这里，我们可以使用 `wait/notify` 实现一套等待/通知的机制：告警发送线程在上报告警消息前必须等待，直到 `AlarmAgent` 与告警服务器的连接成功建立或者恢复；心跳线程在检测到网络连接恢复之后通知告警发送线程，如清单 5-1 所示。

清单 5-1 AlarmAgent 源码

```
/**
 * 告警代理
 *
 * @author Viscent Huang
 */
public class AlarmAgent {
    // 保存该类的唯一实例
    private final static AlarmAgent INSTANCE = new AlarmAgent();
    // 是否连接上告警服务器
    private boolean connectedToServer = false;
    // 心跳线程，用于检测告警代理与告警服务器的网络连接是否正常
    private final HeartbeatThread heartbeatThread = new HeartbeatThread();

    private AlarmAgent() {
        // 什么也不做
    }

    public static AlarmAgent getInstance() {
        return INSTANCE;
    }

    public void init() {
        connectToServer();
        heartbeatThread.setDaemon(true);
        heartbeatThread.start();
    }

    private void connectToServer() {
        // 创建并启动网络连接线程，在该线程中与告警服务器建立连接
        new Thread() {
            @Override
            public void run() {
                doConnect();
            }
        }.start();
    }

    private void doConnect() {
        // 模拟实际操作耗时
        Tools.randomPause(100);
        synchronized (this) {
            connectedToServer = true;
            // 连接已经建立完毕，通知以唤醒告警发送线程
            notify();
        }
    }
}
```

```

public void sendAlarm(String message) throws InterruptedException {
    synchronized (this) {
        // 使当前线程等待，直到告警代理与告警服务器的连接建立完毕或者恢复
        while (!connectedToServer) {
            Debug.info("Alarm agent was not connected to server.");
            wait();
        }
        // 真正将告警消息上报到告警服务器
        doSendAlarm(message);
    }
}

```

```

private void doSendAlarm(String message) {
    // ...
    Debug.info("Alarm sent:%s", message);
}

```

// 心跳线程

```

class HeartbeartThread extends Thread {
    @Override
    public void run() {
        try {
            // 留一定的时间给网络连接线程与告警服务器建立连接
            Thread.sleep(1000);
            while (true) {
                if (checkConnection()) {
                    connectedToServer = true;
                } else {
                    connectedToServer = false;
                    Debug.info("Alarm agent was disconnected from server.");

                    // 检测到连接中断，重新建立连接
                    connectToServer();
                }
                Thread.sleep(2000);
            }
        } catch (InterruptedException e) {
            // 什么也不做;
        }
    }
}

```

// 检测与告警服务器的网络连接情况

```

private boolean checkConnection() {
    boolean isConnected = true;
    final Random random = new Random();

```

// 模拟随机性的网络断链


```

        int rand = random.nextInt(1000);
        if (rand <= 500) {
            isConnected = false;
        }
        return isConnected;
    }
}

```

AlarmAgent 的实例变量 `connectedToServer` 用来表示告警代理与告警服务器的网络连接状态。`sendAlarm` 方法在调用 `doSendAlarm` 方法将告警消息上报到告警服务器之前会先判断 `connectedToServer` 的值。若 `connectedToServer` 值为 `false`（表示网络连接未建立或者已中断），那么告警发送线程会调用 `AlarmAgent.wait()` 来暂停当前线程。这里，告警发送线程就是一个等待线程，布尔表达式“`connectedToServer`”构成了该等待线程的保护条件。心跳线程在检测到网络连接中断的情况下，会调用 `connectToServer` 方法重新创建一个网络连接线程实例来重建网络连接。网络连接线程在其建立与告警服务器的网络连接之后会调用 `AlarmAgent.notify()` 来通知告警发送线程。这里，网络连接线程相当于通知线程。该案例中的等待线程和通知线程是同步在 `AlarmAgent` 实例之上的。

`Object.wait()` 的执行线程会一直处于 `WAITING` 状态，直到通知线程唤醒该线程并且保护条件成立。因此，`Object.wait()` 所实现的等待是无限等待。`Object.wait()` 方法还有个版本，其声明如下：

```
public final void wait(long timeout) throws InterruptedException
```

`Object.wait(long)` 允许我们指定一个超时时间（单位为毫秒）。如果被暂停的等待线程在这个时间内没有被其他线程唤醒，那么 Java 虚拟机会自动唤醒该线程。不过 `Object.wait(long)` 既无返回值也不会抛出特定的异常，以便区分其返回是由于其他线程通知了当前线程还是由于等待超时。因此，使用 `Object.wait(long)` 的时候我们需要一些额外的处理，如清单 5-2 所示。

清单 5-2 使用 `Object.wait(long)` 实现等待超时控制

```

public class TimeoutWaitExample {
    private static final Object lock = new Object();
    private static boolean ready = false;
    protected static final Random random = new Random();

    public static void main(String[] args) throws InterruptedException {
        Thread t = new Thread() {
            @Override
            public void run() {

```

```

for (;;) {
    synchronized (lock) {
        ready = random.nextInt(100) < 20 ? true : false;
        if (ready) {
            lock.notify();
        }
    }

    // 使当前线程暂停一段(随机)时间
    Tools.randomPause(500);
} // for 循环结束
};
t.setDaemon(true);
t.start();
waiter(1000);
}

public static void waiter(final long timeOut) throws InterruptedException {
    if (timeOut < 0) {
        throw new IllegalArgumentException();
    }

    long start = System.currentTimeMillis();
    long waitTime;
    long now;
    synchronized (lock) {
        while (!ready) {
            now = System.currentTimeMillis();
            // 计算剩余等待时间
            waitTime = timeOut - (now - start);
            Debug.info("Remaining time to wait:%sms", waitTime);
            if (waitTime <= 0) {
                // 等待超时退出
                break;
            }
            lock.wait(waitTime);
        } // while 循环结束
    }

    if (ready) {
        // 执行目标动作
        guardedAction();
    } else {
        // 等待超时, 保护条件未成立
        Debug.error("Wait timed out,unable to execution target action!");
    }
}

```

```

    } // 同步块结束
}

private static void guardedAction() {
    Debug.info("Take some action.");
    // ...
}
}

```

在上述代码中，`Object.wait(long)`调用仍要放在一个循环语句之中。在每次调用`Object.wait(long)`之前，我们总是先根据系统当前时间（`now`）和等待方法被调用的时间（`start`）计算出剩余的等待时间（`waitTime`），然后以该时间为参数去调用`Object.wait(long)`。并且，在执行目标动作前我们会再次判断保护条件（`ready==true`）是否成立，此时保护条件若仍然不成立，则说明循环语句中的`Object.wait(long)`的返回是由等待超时导致的。

`Object.wait()`调用相当于`Object.wait(0)`调用。

5.1.2 wait/notify 的开销及问题

下面我们看 `wait/notify` 实现的等待/通知时可能遇到的问题及其解决方法。

- 过早唤醒（`Wakeup too soon`）问题。设一组等待/通知线程同步在对象 `someObject` 之上，如图 5-1 所示，初始状态下所有保护条件均不成立。接着，线程 `N1` 更新了共享变量 `state1` 使得保护条件₁得以成立，此时为了唤醒使用该保护条件的所有等待线程（线程 `W1` 和线程 `W2`），`N1` 执行了 `someObject.notifyAll()`。由于 `someObject.notifyAll()` 唤醒的是 `someObject` 上的所有等待线程，因此这时线程 `W2` 也会被唤醒。然而，`W2` 所使用保护的护条件₂此时并没有成立，这就使得该线程被唤醒之后仍然需要继续等待。这种等待线程在其所需的保护条件并未成立的情况下被唤醒的现象就被称为过早唤醒（`Wakeup too soon`）。过早唤醒使得那些本来无须被唤醒的等待线程也被唤醒了，从而造成资源浪费。这好比你在人群里大喊一声“美女”，便会有许多自我感觉良好的女性回头一样——尽管你要喊的仅仅是其中某一个人，但大家却都以为你是在喊自己。过早唤醒问题可以利用 JDK 1.5 引入的 `java.util.concurrent.locks.Condition` 接口来解决，5.2 节会介绍该接口。

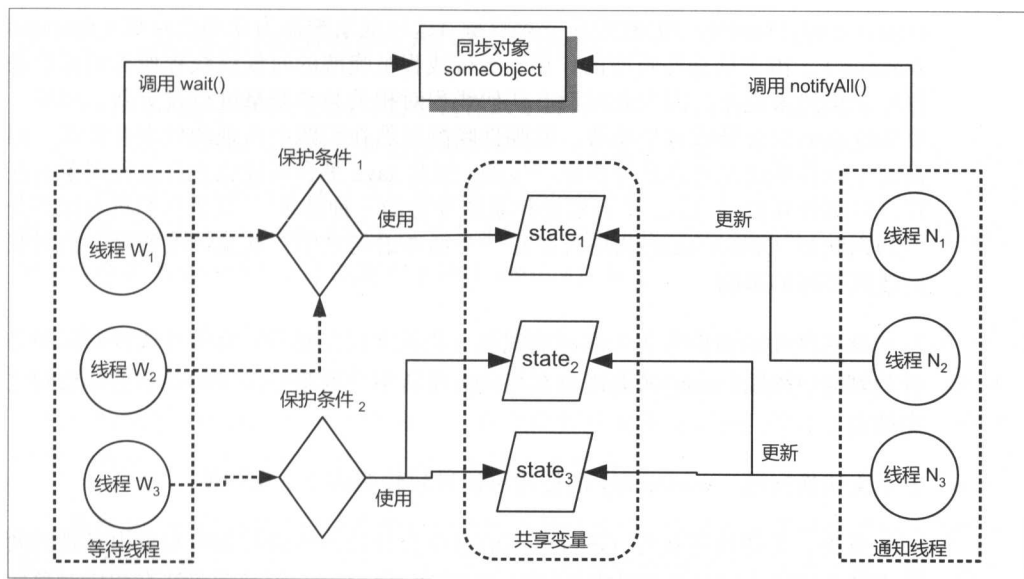


图 5-1 过早唤醒问题示意图

- 信号丢失（Missed Signal）问题。如果等待线程在执行 `Object.wait()` 前没有先判断保护条件是否已然成立，那么有可能出现这种情形——通知线程在该等待线程进入临界区之前就已经更新了相关共享变量，使得相应的保护条件成立并进行了通知，但是此时等待线程还没有被暂停，自然也就无所谓唤醒了。这就可能造成等待线程直接执行 `Object.wait()` 而被暂停的时候，该线程由于没有其他线程进行通知而一直处于等待状态。这种现象就相当于等待线程错过了一个本来“发送”给它的“信号”，因此被称为信号丢失（Missed Signal）。只要将对保护条件的判断和 `Object.wait()` 调用放在一个循环语句之中就可以避免上述场景的信号丢失。信号丢失的另外一个表现是在应该调用 `Object.notifyAll()` 的地方却调用了 `Object.notify()`。比如，对于使用同一个保护条件的多个等待线程，如果通知线程在侦测到这个保护条件成立后调用的是 `Object.notify()`，那么这些等待线程最多只有一个线程能够被唤醒，甚至一个也没有被唤醒——被唤醒的线程是 `Object.notify()` 所属对象上使用其他保护条件的一个等待线程！也就是说，尽管通知线程在调用 `Object.notify()` 前可能考虑（判断）了某个特定的保护条件是否成立，但是 `Object.notify()` 本身在其唤醒线程时是不考虑任何保护条件的！这就可能使得通知线程执行 `Object.notify()` 进行的通知对于使用相应保护条件的等待线程来说丢失了。这种情形下，避免信号丢失的一个方法是在必要的时候使用 `Object.notifyAll()` 来通知。总的来说，信号丢失本质上是一种代码错误，而不是 Java 标准库 API 自身的问题。
- 欺骗性唤醒（Spurious Wakeup）问题。等待线程也可能在没有其他任何线程执行

`Object.notify()/notifyAll()`的情况下被唤醒。这种现象被称为欺骗性唤醒（Spurious Wakeup）。由于欺骗性唤醒的作用，等待线程被唤醒的时候该线程所需的保护条件可能仍然未成立，因为此时没有任何线程对相关共享变量进行过更新。可见，欺骗性唤醒也会导致过早唤醒。欺骗性唤醒虽然在实践中出现的概率非常低，但是由于操作系统是允许这种现象产生的，因此 Java 平台同样也允许这种现象的存在。欺骗性唤醒是 Java 平台对操作系统妥协的一种结果²。只要我们将对保护条件的判断和 `Object.wait()`调用行放在一个循环语句之中，欺骗性唤醒就不会对我们造成实际的影响。

欺骗性唤醒和信号丢失问题的规避方法前文已经提及：将等待线程对保护条件的判断、`Object.wait()`的调用放在相应对象所引导的临界区中的一个循环语句之中即可。

- 上下文切换问题。`wait/notify` 的使用可能导致较多的上下文切换。

首先，等待线程执行 `Object.wait()`至少会导致该线程对相应对象内部锁的两次申请与释放。通知线程在执行 `Object.notify()/notifyAll()`时需要持有相应对象的内部锁，因此 `Object.notify()/notifyAll()`调用会导致一次锁的申请。而锁的申请与释放可能导致上下文切换。

其次，等待线程从被暂停到唤醒这个过程本身就会导致上下文切换。

再次，被唤醒的等待线程在继续运行时需要再次申请相应对象的内部锁，此时等待线程可能需要和相应对象的入口集中的其他线程以及其他新来的活跃线程（即申请相应的内部锁且处于 `RUNNABLE` 状态的线程）争用相应的内部锁，而这又可能导致上下文切换。

最后，过早唤醒问题也会导致额外的上下文切换，这是因为被过早唤醒的线程仍然需要继续等待，即再次经历被暂停和唤醒的过程。

以下方法有助于避免或者减少 `wait/notify` 导致过多的上下文切换。

- 在保证程序正确性的前提下（5.1.3 节会介绍），使用 `Object.notify()`替代 `Object.notifyAll()`。`Object.notify()`调用不会导致过早唤醒，因此减少了相应的上下文切换开销。
- 通知线程在执行完 `Object.notify()/notifyAll()`之后尽快释放相应的内部锁。这样可以避免被唤醒的线程在 `Object.wait()`调用返回前再次申请相应内部锁时，由于该锁尚未被通知线程释放而导致该线程被暂停（以等待再次获得锁的机会）。

2 参见 <http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/Condition.html>。

5.1.3 Object.notify()/notifyAll()的选用

Object.notify()可能导致信号丢失这样的正确性问题,而 Object.notifyAll()虽然效率不太高(把不需要唤醒的等待线程也给唤醒了),但是其在正确性方面有保障。因此实现通知的一种比较流行的保守性方法是优先使用 Object.notifyAll()以保障正确性,只有在有证据表明使用 Object.notify()足够的情况下才使用 Object.notify()——Object.notify()只有在下列条件全部满足的情况下才能够用于替代 notifyAll 方法。

条件 1 一次通知仅需要唤醒至多一个线程。这一点容易理解,但是光满足这一点还不足以用 Object.notify()去替代 Object.notifyAll()。在不同的等待线程可能使用不同的保护条件的情况下, Object.notify()唤醒的一个任意线程可能并不是我们需要唤醒的那一个(种)线程。因此,这个问题还需要通过满足条件 2 来排除。

条件 2 相应对象的等待集中仅包含同质等待线程。所谓同质等待线程指这些线程使用同一个保护条件,并且这些线程在 Object.wait()调用返回之后的处理逻辑一致。最为典型的同质线程是使用同一个 Runnable 接口实例创建的不同线程(实例)或者从同一个 Thread 子类的 new 出来的多个实例。

注意

Object.notify()唤醒的是其所属对象上的一个任意等待线程。Object.notify()本身在唤醒线程时是不考虑保护条件的。Object.notifyAll()方法唤醒的是其所属对象上的所有等待线程。使用 Object.notify()替代 Object.notifyAll()时需要确保以下两个条件同时得以满足:

- 一次通知仅需要唤醒至多一个线程。
- 相应对象上的所有等待线程都是同质等待线程。

*5.1.4 wait/notify 与 Thread.join()

Thread.join()可以使当前线程等待目标线程结束之后才继续运行。Thread.join()还有另外一个如下声明的版本:

```
public final void join(long millis) throws InterruptedException
```

join(long)允许我们指定一个超时时间。如果目标线程没有在指定的时间内终止,那么当前线程也会继续运行。join(long)实际上就是使用了 wait/notify 来实现的,如图 5-2 所示。

```

public final synchronized void join(long millis)
throws InterruptedException {
    long base = System.currentTimeMillis();
    long now = 0;

    if (millis < 0) {
        throw new IllegalArgumentException("timeout value is negative");
    }

    if (millis == 0) {
        while (isAlive()) {
            wait(0);
        }
    } else {
        while (isAlive()) {
            long delay = millis - now;
            if (delay <= 0) {
                break;
            }
            wait(delay);
            now = System.currentTimeMillis() - base;
        }
    }
}

```

图 5-2 Thread.join(long)源码

join(long)是一个同步方法。它检测到目标线程未结束的时候会调用 wait 方法来暂停当前线程，直到目标线程已终止。这里，当前线程相当于等待线程，其所需的保护条件是“目标线程已终止”（Thread.isAlive()返回值为 false）。Java 虚拟机会在目标线程的 run 方法运行结束后执行该线程（对象）的 notifyAll 方法来通知所有的等待线程。可见这里的目标线程充当了同步对象的角色，而 Java 虚拟机中 notifyAll 方法的执行线程则是通知线程。另外，join(long)正是按照清单 5-2 所展示的实现等待超时控制的方法来使用 wait(long)方法的。

Thread.join()调用相当于 Thread.join(0)调用。

5.2 Java 条件变量

总的来说，Object.wait()/notify()过于底层，并且还存在过早唤醒问题以及 Object.wait(long)无法区分其返回是由于等待超时还是被通知线程唤醒等问题。但是，了解 wait/notify 有助于我们理解和维护现有系统，以及学习和使用 JDK 1.5 中引入的新的标准库类 java.util.concurrent.locks.Condition 接口。

Condition 接口可作为 wait/notify 的替代品来实现等待/通知，它为了解决过早唤醒问题

提供了支持,并解决了 `Object.wait(long)` 不能区分其返回是否是由等待超时而导致的问题。`Condition` 接口定义的 `await` 方法、`signal` 方法和 `signalAll` 方法分别相当于 `Object.wait()`、`Object.notify()` 和 `Object.notifyAll()`。

`Lock.newCondition()` 的返回值就是一个 `Condition` 实例,因此调用任意一个显式锁实例的 `newCondition` 方法可以创建一个相应的 `Condition` 接口。`Object.wait()/notify()` 要求其执行线程持有这些方法所属对象的内部锁,类似地, `Condition.await()/signal()` 也要求其执行线程持有创建该 `Condition` 实例的显式锁。`Condition` 实例也被称为条件变量 (Condition Variable) 或者条件队列 (Condition Queue), 每个 `Condition` 实例内部都维护了一个用于存储等待线程的队列 (等待队列)。设 `cond1` 和 `cond2` 是两个不同的 `Condition` 实例,一个线程执行 `cond1.await()` 会导致其被暂停 (线程生命周期状态变更为 `WAITING`) 并被存入 `cond1` 的等待队列。`cond1.signal()` 会使 `cond1` 的等待队列中的一个任意线程被唤醒。`cond1.signalAll()` 会使 `cond1` 的等待队列中的所有线程被唤醒,而 `cond2` 的等待队列中的任何一个等待线程不受此影响。

`Condition` 接口的使用方法与 `wait/notify` 的使用方法相似,如下代码模板所示:

```
class ConditionUsage {
    private final Lock lock = new ReentrantLock();
    private final Condition condition = lock.newCondition();
    public void aGuaredMethod() throws InterruptedException {
        lock.lock();
        try {
            while (保护条件不成立) {
                condition.await();
            }
            // 执行目标动作
            doAction();
        } finally {
            lock.unlock();
        }
    }

    private void doAction() {
        // ...
    }

    public void anNotificationMethod() throws InterruptedException {
        lock.lock();
        try {
            // 更新共享变量
            changeState();
            condition.signal();
        }
    }
}
```



```

        } finally {
            lock.unlock();
        }
    }

    private void changeState() {
        // ...
    }
}

```

可见，`Condition.await()/signal()`的执行线程需要持有创建相应条件变量的显式锁。对保护条件的判断、`Condition.await()`的调用也同样放在一个循环语句之中，并且该循环语句与目标动作的执行放在同一个显式锁所引导的临界区之中，这同样也是考虑到了欺骗性唤醒问题、信号丢失问题。`Condition.await()`与 `Object.wait()`类似，它使当前线程暂停的同时也使当前线程释放其持有的相应显式锁，并且这时 `Condition.await()`调用也同样未返回。被唤醒的等待线程继续运行的时候也需要再次申请相应的显式锁，被唤醒的等待线程再次获得相应的显式锁后 `Condition.await()`调用才返回。上述模板代码中的 `aGuaredMethod` 方法是一个受保护方法，`anNotificationMethod` 方法是一个通知方法。

下面看 `Condition` 接口是如何解决过早唤醒问题的。如果我们改用 `Condition` 接口去实现图 5-1 所示的等待/通知，如图 5-3 所示，情形则有所不同。此时，同步对象 `someObject` 内部可以维护两个条件变量：`cond1` 和 `cond2`。由于 3 个等待线程所使用的保护条件并不完全相同，因此我们可以使等待线程 W_1 和等待线程 W_2 调用 `cond1.await()`来实现其等待，而等待线程 W_3 则调用 `cond2.await()`来实现其等待。当通知线程更新了状态变量 `state1`之后，该线程只需要调用 `cond1.signalAll()`来唤醒 `cond1` 等待队列中的所有等待线程（ W_1 和 W_2 ）即可。由于线程 W_3 进行等待的时候调用的是另外一个条件变量（`cond2`）的 `await` 的方法，它进入的是 `cond2` 的等待队列，因此通知线程执行 `cond1.signalAll()`并不会使 W_3 被唤醒。可见，使用 `Condition` 接口之后通知线程在更新 `state1`后所唤醒的等待线程只有 W_1 和 W_2 而等待线程 W_3 并不会受此影响，即避免了等待线程 W_3 被过早地唤醒。

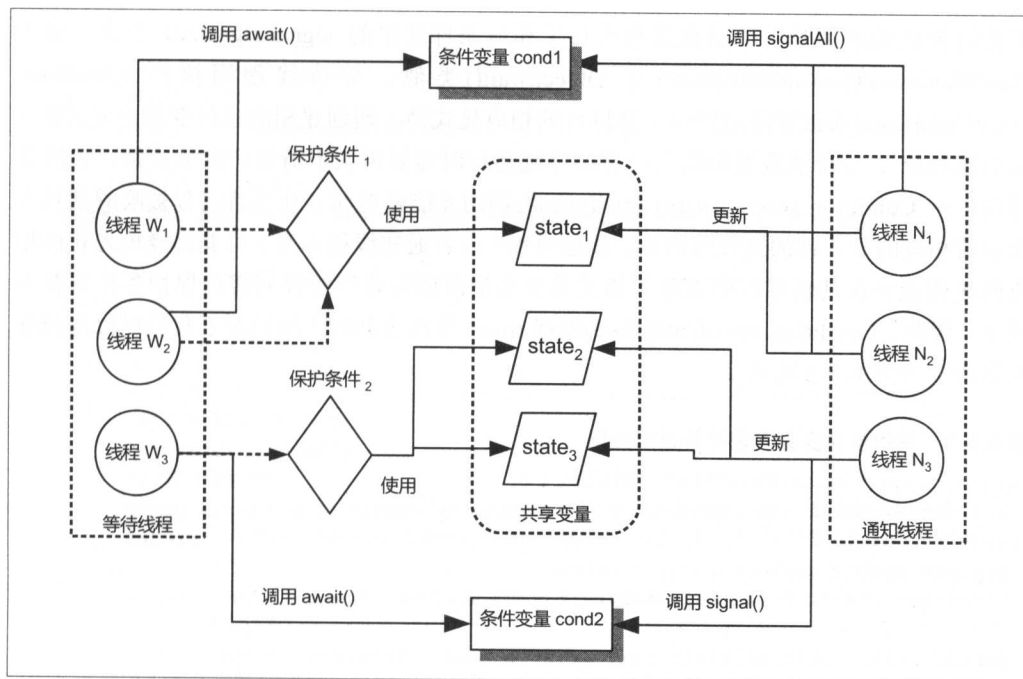


图 5-3 利用 Condition 接口规避过早唤醒问题示意图

可见，应用代码是这样解决过早唤醒问题的：在应用代码这一层次上建立保护条件与条件变量之间的对应关系，即让使用不同保护条件的等待线程调用不同的条件变量的 `await` 方法来实现其等待；并让通知线程在更新了共享变量之后，仅调用涉及了这些共享变量的保护条件所对应的条件变量的 `signal/signalAll` 方法来实现通知。

注意 Condition 接口本身只是对解决过早唤醒问题提供了支持。要真正解决过早唤醒问题，我们需要通过应用代码维护保护条件与条件变量之间的对应关系，即使用不同的保护条件的等待线程需要调用不同的条件变量的 `await` 方法来实现其等待，并使通知线程在更新了相关共享变量之后，仅调用与这些共享变量有关的保护条件所对应的条件变量的 `signal/signalAll` 方法来实现通知。

Condition 接口还解决了 `Object.wait(long)` 存在的问题——`Object.wait(long)` 无法区分其返回是由于等待超时还是被通知的。`Condition.awaitUntil(Date deadline)` 可以用于实现带超时间限制的等待，并且该方法的返回值能够区分该方法调用是由于等待超时而返回还是由于其他线程执行了相应条件变量的 `signal/signalAll` 方法而返回。`Condition.awaitUntil(Date deadline)` 的唯一参数 `deadline` 表示等待的最后期限（Deadline），过了这个时间点就算等待超时。`Condition.awaitUntil(Date)` 返回值 `true` 表示进行的等待尚未达到最后期限，

即此时方法的返回是由于其他线程执行了相应条件变量的 `signal/signalAll` 方法。由于 `Condition.await()/awaitUntil(Date)` 与 `Object.wait()` 类似，等待线程因执行 `Condition.awaitUntil(Date)` 而被暂停的同时，其持有的相应显式锁（即创建相应条件变量的显式锁）也会被释放³，等待线程被唤醒之后得以继续运行时需要再次申请相应的显式锁，然后等待线程对 `Condition.await()/awaitUntil(Date)` 的调用才能够返回。在等待线程被唤醒到其再次申请相应的显式锁的这段时间内，其他线程（或者通知线程本身）可能已经抢先获得相应的显式锁并在其临界区中更新了相关共享变量而使得等待线程所需的保护条件重新不成立。因此，`Condition.awaitUntil(Date)` 返回 `true`（等待未超时）的情况下我们可以选择继续等待，如清单 5-3 所示。

清单 5-3 使用条件变量实现等待超时控制

```
public class TimeoutWaitWithCondition {
    private static final Lock lock = new ReentrantLock();
    private static final Condition condition = lock.newCondition();
    private static boolean ready = false;
    protected static final Random random = new Random();

    public static void main(String[] args) throws InterruptedException {
        Thread t = new Thread() {
            @Override
            public void run() {
                for (;;) {
                    lock.lock();
                    try {
                        ready = random.nextInt(100) < 5 ? true : false;
                        if (ready) {
                            condition.signal();
                        }
                    } finally {
                        lock.unlock();
                    }

                    // 使当前线程暂停一段（随机）时间
                    Tools.randomPause(500);
                } // for 循环结束
            }
        };
        t.setDaemon(true);
        t.start();
        waiter(1000);
    }
}
```

3 即线程的暂停与显式锁的释放是一个原子操作。

```

public static void waiter(final long timeOut) throws InterruptedException {
    if (timeOut < 0) {
        throw new IllegalArgumentException();
    }
    // 计算等待的最后期限
    final Date deadline = new Date(System.currentTimeMillis() + timeOut);
    // 是否继续等待
    boolean continueToWait = true;
    lock.lock();
    try {
        while (!ready) {
            Debug.info("still not ready, continue to wait:%s", continueToWait);
            // 等待未超时, 继续等待
            if (!continueToWait) {
                // 等待超时退出
                Debug.error("Wait timed out, unable to execution target action!");
                return;
            }
            continueToWait = condition.awaitUntil(deadline);
        } // while 循环结束

        // 执行目标动作
        guardedAction();
    } finally {
        lock.unlock();
    }
}

private static void guardedAction() {
    Debug.info("Take some action.");
    // ...
}
}

```

在上述代码中, 我们根据系统当前时间和等待超时时间限制 (timeOut) 来计算出等待的最后期限 (deadline), 并以此为参数去调用 Condition.awaitUntil(Date)。这里 Condition.awaitUntil(Date)调用与 Condition.await()调用一样, 也要放在一个循环语句之中。如果 Condition.awaitUntil(Date)调用返回 false (表示等待超时), 那么等待方法就直接返回, 否则等待方法可以继续等待。

使用条件变量所产生的开销与 wait/notify 方法基本相似; 不过由于条件变量的使用可以避免过早唤醒问题, 因此其使用导致的上下文切换要比 wait/notify 少一些。

5.3 倒计时协调器：CountDownLatch

`Thread.join()`实现的是一个线程等待另外一个线程结束。有时候一个线程可能只需要等待其他线程执行的特定操作结束即可，而不必等待这些线程终止。当然，此时我们也可以使用条件变量来实现。不过，此时我们可以使用更加直接的工具类——`java.util.concurrent.CountDownLatch`。

`CountDownLatch` 可以用来实现一个（或者多个）线程等待其他线程完成一组特定的操作之后才继续运行。这组操作被称为先决操作。

`CountDownLatch` 内部会维护一个用于表示未完成的先决操作数量的计数器。`CountDownLatch.countDown()` 每被执行一次就会使相应实例的计数器值减少 1。`CountDownLatch.await()` 相当于一个受保护方法，其保护条件为“计数器值为 0”（代表所有先决操作已执行完毕），目标操作是一个空操作。因此，当计数器值不为 0 时 `CountDownLatch.await()` 的执行线程会被暂停，这些线程就被称为相应 `CountDownLatch` 上的等待线程。`CountDownLatch.countDown()` 相当于一个通知方法，它会在计数器值达到 0 的时候唤醒相应实例上的所有等待线程。计数器的初始值是在 `CountDownLatch` 的构造参数中指定的，如下声明所示：

```
public CountDownLatch(int count)
```

`count` 参数用于表示先决操作的数量或者需要被执行的次数。当计数器的值达到 0 之后，该计数器的值就不再发生变化。此时，调用 `CountDownLatch.countDown()` 并不会导致异常的抛出，并且后续执行 `CountDownLatch.await()` 的线程也不会被暂停。因此，`CountDownLatch` 的使用是一次性的：一个 `CountDownLatch` 实例只能实现一次等待和唤醒。

可见，`CountDownLatch` 内部封装了对“全部先决操作已执行完毕”（计数器值为 0）这个保护条件的等待与通知的逻辑，因此客户端代码在使用 `CountDownLatch` 实现等待/通知的时候调用 `await`、`countDown` 方法都无须加锁。

下面看一个实战案例。某定制 Web 服务器（以下称其为服务器）在其启动时需要启动若干启动过程比较耗时的服务（以下称其为服务）。为了尽可能地减少服务器启动过程的总耗时，该服务器会使用专门的工作者线程以并发的方式去启动这些服务。但是，服务器在所有启动操作结束后，需要判断这些服务的状态以检查服务器的启动是否是成功的。只有在这些服务全部启动成功的情况下该服务器才被认为是启动成功的，否则服务器的启动算失败，此时服务器会自动终止（退出 Java 虚拟机），如清单 5-4 所示。

清单 5-4 服务器启动代码

```

public class ServerStarter {

    public static void main(String[] args) {
        // 省略其他代码

        // 启动所有服务
        ServiceManager.startServices();

        // 执行其他操作

        // 在所有其他操作执行结束后，检测服务启动状态
        boolean allIsOK;
        // 检测全部服务的启动状态
        allIsOK = ServiceManager.checkServiceStatus();

        if (allIsOK) {
            System.out.println("All services were sucessfully started!");
            // 省略其他代码
        } else {
            // 个别服务启动失败，退出 JVM
            System.err.println("Some service(s) failed to start, exiting JVM...");
            System.exit(1);
        }
        // ...
    }
}

```

这里，`ServiceManager.startServices()`负责启动所有服务。其中，每个服务都有一个相应的服务启动线程负责执行该服务的启动操作。`ServiceManager.checkServiceStatus()`用于检查全部服务的启动状态，该方法的执行线程为 `main` 线程。显然，我们只有在所有服务的启动操作执行结束后才能够检查这些服务的启动状态（最终状态）。即检查服务启动状态这个操作的先决操作是所有服务启动动作执行完毕。因此，我们可以创建一个 `CountDownLatch` 实例 `latch`，`ServiceManager.checkServiceStatus()`会调用 `latch.await()`进行等待，直到所有服务的启动动作被各自的服务启动线程执行完毕。每个服务启动线程在执行完相应服务的启动动作后都会执行 `latch.countDown()`。`ServiceManager` 的源码如清单 5-5 所示。

清单 5-5 `ServiceManager` 源码

```

public class ServiceManager {
    static volatile CountDownLatch latch;
    static Set<Service> services;
}

```

```
public static void startServices() {
    services = getServices();
    for (Service service : services) {
        service.start();
    }
}

public static boolean checkServiceStatus() {
    boolean allIsOK = true;
    // 等待服务启动结束
    try {
        latch.await();
    } catch (InterruptedException e) {
        return false;
    }

    for (Service service : services) {
        if (!service.isStarted()) {
            allIsOK = false;
            break;
        }
    }
    return allIsOK;
}

static Set<Service> getServices() {
    // 模拟实际代码
    latch = new CountDownLatch(3);
    HashSet<Service> servcies = new HashSet<Service>();
    servcies.add(new SampleServiceC(latch));
    servcies.add(new SampleServiceA(latch));
    servcies.add(new SampleServiceB(latch));
    return servcies;
}
```

服务启动的公共处理逻辑在 Service 接口的抽象实现类 AbstractService 中定义，如清单 5-6 所示。

清单 5-6 AbstractService 源码

```
public abstract class AbstractService implements Service {
    protected boolean started = false;
    protected final CountDownLatch latch;

    public AbstractService(CountDownLatch latch) {
        this.latch = latch;
    }
}
```

```

@Override
public boolean isStarted() {
    return started;
}

// 留给子类实现的抽象方法，用于实现服务器的启动逻辑
protected abstract void doStart() throws Exception;

@Override
public void start() {
    new ServiceStarter().start();
}

@Override
public void stop() {
    // 默认什么也不做
}

class ServiceStarter extends Thread {
    @Override
    public void run() {
        final String serviceName = AbstractService.this.getClass()
            .getSimpleName();
        Debug.info("Starting %s", serviceName);
        try {
            doStart();
            started = true;
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            latch.countDown();
            Debug.info("Done Starting %s", serviceName);
        }
    }
}

```

AbstractService 为每个服务创建了一个服务启动线程 (ServiceStarter 实例)。服务启动线程会执行 AbstractService 子类的 doStart 方法中实现的服务启动动作，并在该方法返回后将相应服务的状态设置为“已启动”。doStart 方法调用结束之后，不管其成功返回（说明服务启动成功）还是抛出异常（说明服务启动失败），服务启动线程都会执行 latch.countDown()。

从清单 5-6 可以看出，服务启动线程执行 latch.countDown() 所起到的作用仅相当于向

CountDownLatch 实例报告它完成了某个操作，而 CountDownLatch.countDown()本身是无法区分被报告操作的结果是成功的还是失败的。这就是 ServiceStarter 类（见清单 5-6）的 run 方法中 countDown 方法是在 finally 块中调用的原因。为了能够标识相应服务的启动状态，我们在 AbstractService 类中维护一个实例变量 started。

如果 CountDownLatch 内部计数器由于程序的错误而永远无法达到 0，那么相应实例上的等待线程会一直处于 WAITING 状态。避免该问题的出现有两种方法。

其一，确保所有 CountDownLatch.countDown()调用都位于代码中正确的位置。例如本案例，如果我们把 CountDownLatch.countDown()调用放在 doStart()调用之后而不是 finally 块中（见清单 5-6），那么某个服务启动过程中出现异常（如运行时异常）会导致 main 线程执行到 ServiceManager.checkServiceStatus()时，该线程一直处于 WAITING 状态。

其二，等待线程在等待先决操作完成的时候指定一个时间限制。此时我们可以使用 CountDownLatch.await()的另外一个版本，其声明如下：

```
public boolean await(long timeout, TimeUnit unit)
                    throws InterruptedException
```

CountDownLatch.await(long, TimeUnit)允许指定一个超时时间，在该时间内如果相应 CountDownLatch 实例的计数器值仍然未达到 0，那么所有执行该实例的 await 方法的线程都会被唤醒。该方法的返回值可用于区分其返回是否是由于等待超时。

注意

CountDownLatch 内部计数器值达到 0 后其值就恒定不变，后续执行该 CountDownLatch 实例的 await 方法的任何一个线程都不会被暂停。为了避免等待线程永远被暂停，CountDownLatch.countDown()调用必须放在代码中总是可以被执行到的地方，例如 finally 块中。

对于同一个 CountDownLatch 实例 latch，latch.countDown()的执行线程在执行该方法之前所执行的任何内存操作对等待线程在 latch.await()调用返回之后的代码是可见的且有序的。

前文我们说过 CountDownLatch 的构造器中的参数既可以表示先决操作的数量，也可以表示先决操作需要被执行的次数。在上述实战案例中，CountDownLatch 的构造器中的参数的含义就属于前者。而后者表示我们可以在一个线程中多次调用同一个 CountDownLatch 实例的 countDown 方法，以使相应实例的内部计数器值达到 0，如清单 5-7 所示。

清单 5-7 一个线程多次执行 `CountDownLatch.countDown()` 示例

```

public class CountDownLatchExample {
    private static final CountDownLatch latch = new CountDownLatch(4);
    private static int data;

    public static void main(String[] args) throws InterruptedException {
        Thread workerThread = new Thread() {
            @Override
            public void run() {
                for (int i = 1; i < 10; i++) {
                    data = i;
                    latch.countDown();
                    // 使当前线程暂停（随机）一段时间
                    Tools.randomPause(1000);
                }
            };
        };
        workerThread.start();
        latch.await();
        Debug.info("It's done. data=%d", data);
    }
}

```

我们在创建 `CountDownLatch` 实例 `latch` 的时候指定的构造器参数为 4。尽管 `latch.countDown()` 一共会被子线程 `workerThread` 执行 10 次，但是该程序的输出总是如下：

```
It's done. data=4
```

这里程序输出的 `data` 为 4 而不是 10 是由于：首先，`latch.countDown()` 被 `workerThread` 执行了 4 次之后，`main` 线程对 `latch.await()` 的调用就返回了，从而使该线程被唤醒。其次，`workerThread` 在执行 `latch.countDown()` 前所执行的操作（更新共享变量 `data`）的结果对等待线程（`main` 线程）从 `latch.await()` 返回之后的代码可见，因此 `main` 线程被唤醒时能够读取到此前 `workerThread` 在 `latch.countDown()` 调用返回前的操作结果——`data` 被更新为 4。

这里，`latch.countDown()` 被 `workerThread` 执行的次数大于 4 次并不会导致异常，也不会导致 `latch` 内部状态（计数器值）的变更。

5.4 栅栏（CyclicBarrier）

有时候多个线程可能需要相互等待对方执行到代码中的某个地方（集合点），这时这些线程才能够继续执行。这种等待类似于大家相约去爬山的情形：大家事先约定好时间和集合点，先到的人必须在集合点等待其他未到的人，只有所有参与人员到齐之后大家才能

够出发去登山。JDK 1.5 开始引入了一个类 `java.util.concurrent.CyclicBarrier`，该类可以用来实现这种等待。`CyclicBarrier` 类的类名中虽然包含 `Barrier` 这个单词，但是它和我们前面讲的内存屏障没有直接的关联。类名中 `Cyclic` 表示 `CyclicBarrier` 实例是可以重复使用的。

使用 `CyclicBarrier` 实现等待的线程被称为参与方（Party）。参与方只需要执行 `CyclicBarrier.await()` 就可以实现等待。尽管从应用代码的角度来看，参与方是并发执行 `CyclicBarrier.await()` 的，但是，`CyclicBarrier` 内部维护了一个显式锁，这使得其总是可以在所有参与方中区分出一个最后执行 `CyclicBarrier.await()` 的线程，该线程被称为最后一个线程。除最后一个线程外的任何参与方执行 `CyclicBarrier.await()` 都会导致该线程被暂停（线程生命周期状态变为 `WAITING`）。最后一个线程执行 `CyclicBarrier.await()` 会使得使用相应 `CyclicBarrier` 实例的其他所有参与方被唤醒，而最后一个线程自身并不会被暂停。如果把参与方比作爬山例子中的登山者，那么参与方执行 `CyclicBarrier.await()` 而被暂停就相当于登山者到达指定的集合点等待其他登山者；最后一个线程执行 `CyclicBarrier.await()` 则相当于最后一个登山者到达集合点，这使得所有登山者都无须继续等待（被唤醒）而是可以一起出发去爬山（线程继续运行）。与 `CountDownLatch` 不同的是，`CyclicBarrier` 实例是可重复使用的：所有参与方被唤醒的时候，任何线程再次执行 `CyclicBarrier.await()` 又会被暂停，直到这些线程中的最后一个线程执行了 `CyclicBarrier.await()`。

下面看一个例子。该例子模拟了士兵参与打靶训练。所有参与训练的士兵（`Soldier`）被分为若干组（`Rank`），其中每组被称为一排。一排中士兵的个数等于靶子的个数。每次只能够有一排士兵进行射击。一排中的士兵必须同时开始射击，并且射击完毕的士兵必须等待同排的其他所有士兵射击完毕后才能整排地撤离射击点。一排中的士兵射击结束后腾出射击点和靶子，换另外一排中的士兵进行下一轮射击，如此交替进行，直到训练时间结束，如清单 5-8 所示。

清单 5-8 `CyclicBarrier` 使用示例

```
public class ShootPractice {
    // 参与打靶训练的全部士兵
    final Soldier[][] rank;
    // 靶的个数，即每排中士兵的个数
    final int N;
    // 打靶持续时间（单位：秒）
    final int lasting;
    // 标识是否继续打靶
    volatile boolean done = false;
    // 用来指示进行下一轮打靶的是哪一排的士兵
    volatile int nextLine = 0;
    final CyclicBarrier shiftBarrier;
    final CyclicBarrier startBarrier;
```

```

public ShootPractice(int N, final int lineCount, int lasting) {
    this.N = N;
    this.lasting = lasting;
    this.rank = new Soldier[lineCount][N];
    for (int i = 0; i < lineCount; i++) {
        for (int j = 0; j < N; j++) {
            rank[i][j] = new Soldier(i * N + j);
        }
    }
    shiftBarrier = new CyclicBarrier(N, new Runnable() {
        @Override
        public void run() {
            // 更新新一轮打靶的排
            nextLine = (nextLine + 1) % lineCount; // 语句①
            Debug.info("Next turn is :%d", nextLine);
        }
    });
    // 语句②
    startBarrier = new CyclicBarrier(N);
}

public static void main(String[] args) throws InterruptedException {
    ShootPractice sp = new ShootPractice(4, 5, 24);
    sp.start();
}

public void start() throws InterruptedException {
    // 创建并启动工作者线程
    Thread[] threads = new Thread[N];
    for (int i = 0; i < N; ++i) {
        threads[i] = new Shooting(i);
        threads[i].start();
    }
    // 指定时间后停止打靶
    Thread.sleep(lasting * 1000);
    stop();
    for (Thread t : threads) {
        t.join();
    }
    Debug.info("Practice finished.");
}

public void stop() {
    done = true;
}

class Shooting extends Thread {

```

```

final int index;

public Shooting(int index) {
    this.index = index;
}

@Override
public void run() {
    Soldier soldier;
    try {
        while (!done) {
            soldier = rank[nextLine][index];
            // 一排中的士兵必须同时开始射击
            startBarrier.await(); // 语句③
            // 该士兵开始射击
            soldier.fire();
            // 一排中的士兵必须等待该排中的所有其他士兵射击完毕才能够离开射击点
            shiftBarrier.await(); // 语句④
        }
    } catch (InterruptedException e) {
        // 什么也不做
    } catch (BrokenBarrierException e) {
        e.printStackTrace();
    }
}

// run 方法结束
} // 类 Shooting 定义结束

// 参与打靶训练的士兵
static class Soldier {
    private final int seqNo;
    public void fire() {
        // 完整代码见本书配套下载资源
    }
    // 完整代码见本书配套下载资源
} // 类 Soldier 定义结束
}

```

我们使用了两个 `CyclicBarrier` 实例：`startBarrier` 和 `shiftBarrier`。其中 `startBarrier` 用于实现当前排的士兵在同一时刻开始射击，`shiftBarrier` 用于实现当前排的士兵在该排所有士兵射击完毕后同时撤离打靶位置。由于一排中的士兵必须同时开始射击，因此一排中的任意一个士兵在其开始射击前必须等待同排中的其他士兵准备就绪，等到该排中所有士兵准备就绪的时候，该排中的所有士兵都开始射击。这种等待的模拟是通过执行 `startBarrier.await()` 实现的（见语句③）。虽然一排中的士兵是同时开始射击的，但是由于不同士兵的熟练程度不同，因此他们的射击结束时间是不同的。一排中先射击完毕的士兵

必须等待同排中的其他士兵都射击完毕后才能撤离射击点。因此，先射击完毕的士兵需要原地等待，这种等待的模拟是通过调用 `shiftBarrier.await()` 实现的（见语句④）。另外，一排士兵射击结束后撤离射击点时下一排士兵可以进入射击点。这种打靶轮次的转换模拟是在语句①中实现的。`CyclicBarrier` 的其中一个构造器允许我们指定一个被称为 `barrierAction` 的任务（`Runnable` 接口实例）。`barrierAction` 会被最后一个线程执行 `CyclicBarrier.await` 方法时执行，该任务执行结束后其他等待线程才会被唤醒。语句①就是利用了这一点而被放到了 `barrierAction.run()` 之中执行，从而确保了一排士兵射击结束后，下一排进行射击的士兵也随之确定了。

该例子的运行输出类似如下（省略部分输出）：

```
[2016-04-18 23:01:05.063][INFO][Thread-1]:Soldier-1 start firing...
[2016-04-18 23:01:05.062][INFO][Thread-0]:Soldier-0 start firing...
[2016-04-18 23:01:05.062][INFO][Thread-2]:Soldier-2 start firing...
[2016-04-18 23:01:05.062][INFO][Thread-3]:Soldier-3 start firing...
Soldier-2 fired.
Soldier-3 fired.
Soldier-0 fired.
Soldier-1 fired.
[2016-04-18 23:01:09.812][INFO][Thread-1]:Next turn is :1
[2016-04-18 23:01:09.812][INFO][Thread-0]:Soldier-4 start firing...
[2016-04-18 23:01:09.812][INFO][Thread-2]:Soldier-6 start firing...
[2016-04-18 23:01:09.812][INFO][Thread-1]:Soldier-5 start firing...
[2016-04-18 23:01:09.813][INFO][Thread-3]:Soldier-7 start firing...
Soldier-4 fired.
Soldier-5 fired.
Soldier-6 fired.
Soldier-7 fired.
[2016-04-18 23:01:14.164][INFO][Thread-3]:Next turn is :2
[2016-04-18 23:01:14.164][INFO][Thread-3]:Soldier-11 start firing...
[2016-04-18 23:01:14.164][INFO][Thread-2]:Soldier-10 start firing...
[2016-04-18 23:01:14.165][INFO][Thread-0]:Soldier-8 start firing...
[2016-04-18 23:01:14.165][INFO][Thread-1]:Soldier-9 start firing...
...省略其他输出...

[2016-04-18 23:01:27.815][INFO][Thread-0]:Next turn is :0
[2016-04-18 23:01:27.815][INFO][Thread-1]:Soldier-1 start firing...
[2016-04-18 23:01:27.815][INFO][Thread-0]:Soldier-0 start firing...
[2016-04-18 23:01:27.815][INFO][Thread-2]:Soldier-2 start firing...
[2016-04-18 23:01:27.815][INFO][Thread-3]:Soldier-3 start firing...
Soldier-1 fired.
Soldier-2 fired.
Soldier-3 fired.
Soldier-0 fired.
```

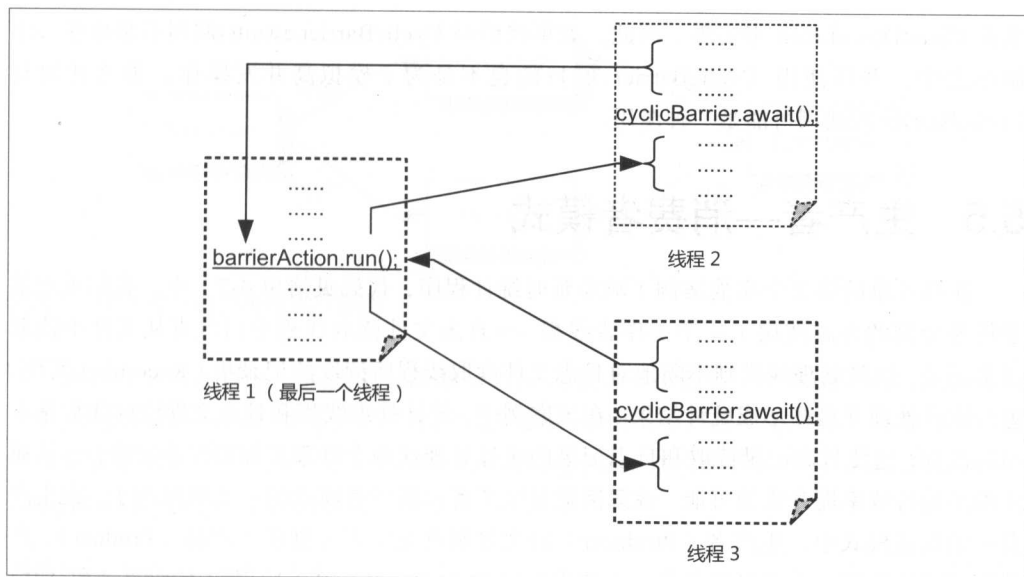
从以上输出可以看出：其一，一排中的士兵开始射击的时间点非常接近（时间相差仅几毫秒）；其二，执行语句①的线程总是模拟最后一个结束射击的士兵的工作者线程（最后一个线程）；其三，进行射击的排次是从 0 开始循环递增的。

由于 `CyclicBarrier` 内部实现是基于条件变量的，因此 `CyclicBarrier` 的开销与条件变量的开销相似，其主要开销在可能产生的上下文切换。

扩展阅读 `CyclicBarrier` 的内部实现

`CyclicBarrier` 内部使用了一个条件变量 `trip` 来实现等待/通知。`CyclicBarrier` 内部实现使用了分代（`Generation`）的概念用于表示 `CyclicBarrier` 实例是可以重复使用的。除最后一个线程外的任何一个参与方都相当于一个等待线程，这些线程所使用的保护条件是“当前分代内，尚未执行 `await` 方法的参与方个数（`parties`）为 0”。当前分代的初始状态是 `parties` 等于参与方总数（通过构造器中的 `parties` 参数指定）。`CyclicBarrier.await()` 每被执行一次会使相应实例的 `parties` 值减少 1。最后一个线程相当于通知线程，它执行 `CyclicBarrier.await()` 会使相应实例的 `parties` 值变为 0，此时该线程会先执行 `barrierAction.run()`，然后再执行 `trip.signalAll()` 来唤醒所有等待线程。接着，开始下一个分代，即使得 `CyclicBarrier` 的 `parties` 值又重新恢复为其初始值。

设 `cyclicBarrier` 为一个任意的 `CyclicBarrier` 实例，任意一个参与方在执行 `cyclicBarrier.await()` 前所执行的任何操作对 `barrierAction.run()` 而言是可见的、有序的。`barrierAction.run()` 中所执行的任何操作对所有参与方在 `cyclicBarrier.await()` 调用成功返回之后的代码而言是可见的、有序的。如图 5-4 所示（图中实线表示其一端连接的操作对箭头指向的代码是可见的、有序的）。

图 5-4 `CyclicBarrier` 中的可见性、有序性保障

CyclicBarrier 的典型应用场景

`CyclicBarrier` 的典型应用场景包括以下几个，它们都可以在上述例子中找到影子。

- 使迭代 (Iterative) 算法并发化。在并发化的迭代算法中，迭代操作是由多个工作者线程并行执行的。`CyclicBarrier` 可用来实现执行迭代操作的任何一个工作者线程必须等待其他工作者线程也完成当前迭代操作的情况下才继续其下一轮的迭代操作，以便形成迭代操作的中间结果作为下一轮迭代的基础 (输入)。因此，该应用场景从代码上反映出来的是，`CyclicBarrier.await()` 调用是在一个循环中执行的，正如清单 5-8 中语句④所在的循环语句所展示的那样。
- 在测试代码中模拟高并发。在编写多线程程序的测试代码时，我们常常需要使用有限的工作者线程来模拟高并发操作。为此，`CyclicBarrier` 可用来实现这些工作者线程中的任意一个线程在执行其操作前必须等待其他线程也准备就绪，即使得这些工作者线程尽可能地在同一时刻开始其操作，正如上述例子我们使用 `CyclicBarrier` 来实现一排中的士兵在同一时刻开始射击。

`CyclicBarrier` 往往被滥用，其表现是在没有必要使用 `CyclicBarrier` 的情况下使用了 `CyclicBarrier`。这种滥用的一个典型例子是利用 `CyclicBarrier` 的构造器参数 `barrierAction` 来指定一个任务，以实现一种等待线程结束的效果：`barrierAction` 中的任务只有在目标线程结束后才能够被执行。事实上，这种情形下我们完全可以使用更加对口的 `Thread.join()`

或者 `CountDownLatch` 来实现。因此，如果代码对 `CyclicBarrier.await()` 调用不是放在一个循环之中，并且使用 `CyclicBarrier` 的目的也不是为了模拟高并发操作，那么此时对 `CyclicBarrier` 的使用可能是一种滥用。

5.5 生产者—消费者模式

在第 4 章的第 2 个实战案例（响应延时统计程序，代码见清单 4-7）中，我们采用基于任务分割的方式使用了一个工作者线程——日志文件读取线程专门负责从文件中读取日志记录。统计处理线程则不断地对日志文件读取线程所读取的记录集（`RecordSet` 实例）进行统计处理并最终形成统计结果。在该案例中，统计处理线程和日志文件读取线程是不同的线程，这使日志记录读取和日志记录的统计处理这两个处理步骤得以并发进行，从而使程序运行效率提升成为可能。该案例就是生产者—消费者模式的一个典型例子。在生产者—消费者模式中，生产者（`Producer`）的主要职责是生产（创建）产品（`Product`）。产品既可以是数据，也可以是任务。上述案例中的 `AbstractLogReader` 类（代码见清单 4-9）就相当于生产者（`Producer`），该类负责读取日志文件并将其读取到的一批日志记录填充到指定的记录集。这里 `AbstractLogReader` 类所填充的记录集就相当于产品。相应地，通过 `AbstractLogReader` 类创建的线程（日志文件读取线程 `logReaderThread`）就被称为生产者线程。消费者（`Consumer`）的主要职责是消费生产者所生产的产品。这里的“消费”包括对产品所代表的数据进行加工处理或者执行产品所代表的任务。上述案例中的 `MultithreadedStatTask` 类（代码见清单 4-7）就相当于消费者，`MultithreadedStatTask.doCalculate()` 对生产者所生产的记录集（产品）进行统计处理（消费）。而 `MultithreadedStatTask.doCalculate()` 是在 `main` 线程（统计处理线程）中执行的，因此这里的 `main` 线程被相应地称为消费者线程。可见，生产者和消费者是并发地运行在各自的线程之中的，这就意味着运用生产者—消费者模式可以使程序中原本串行的处理得以并发化。例如上述案例生产者、消费者相互协作使得日志记录读取和日志记录统计处理这两个处理步骤得以并发化。

由于线程之间无法像函数调用那样通过参数直接传递数据，因此生产者和消费者之间需要一个用于传递产品的传输通道（`Channel`）。传输通道相当于生产者和消费者之间的缓冲区，生产者每生产一个产品就将其放入传输通道，消费者则不断地从传输通道中取出产品进行消费，传输通道通常可以使用一个线程安全的队列来实现。生产者—消费者模式如图 5-5 所示，其中 `Producer` 可以运行在一个或者多个线程中，`Consumer` 也可以运行在一个或者多个线程中。

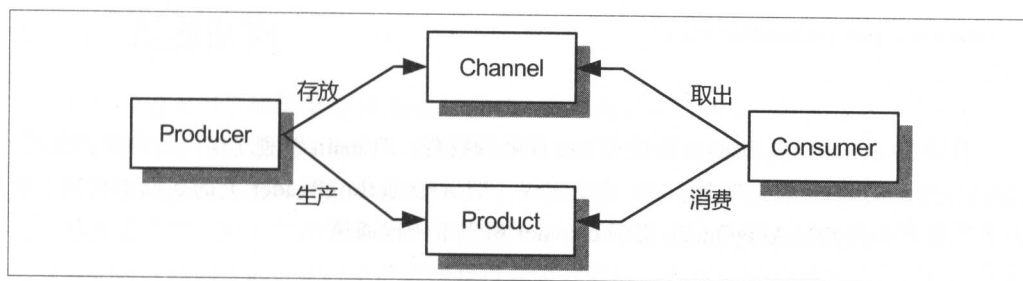


图 5-5 生产者—消费者模式示意图

例如在上述案例中，MultithreadedStatTask 类相当于消费者，AbstractLogReader 类的子类 LogReaderThread 相当于生产者，RecordSet 类相当于产品。生产者、消费者之间通过线程安全的队列——java.util.concurrent.BlockingQueue 接口的实现类 java.util.concurrent.ArrayBlockingQueue 来传递数据，如清单 5-9 所示。

清单 5-9 生产者—消费者实例

```
/**
 * 日志读取线程实现类
 *
 * @author Viscent Huang
 */
public class LogReaderThread extends AbstractLogReader {
    // 线程安全的队列
    final BlockingQueue<RecordSet> channel = new ArrayBlockingQueue<RecordSet>(2);

    public LogReaderThread(InputStream in, int inputBufferSize, int batchSize) {
        super(in, inputBufferSize, batchSize);
    }

    @Override
    public RecordSet nextBatch()
        throws InterruptedException {
        RecordSet batch;
        // 从队列中取出一个记录集
        batch = channel.take();
        if (batch.isEmpty()) {
            batch = null;
        }
        return batch;
    }

    @Override
    protected void publish(RecordSet recordBatch) throws InterruptedException {
        // 记录集存入队列
    }
}
```

```
channel.put(recordBatch);  
}  
}
```

其中，nextBatch 方法的执行线程（统计处理线程，即 main 线程）相当于消费者线程；publish 方法的执行线程（日志文件读取线程，即 AbstractLogReader 类的实例）相当于生产者线程；ArrayBlockingQueue 实例 channel 相当于传输通道。

术语定义 将产品存入传输通道的线程就被称为生产者线程，从传输通道中取出产品进行消费的线程就被称为消费者线程。

由于生产者和消费者运行在不同的线程中，因此生产者将产品（对象）存入传输通道，消费者再从相应的传输通道中取出产品的过程其实就是生产者线程将对象（产品）发布到消费者线程的过程，这种对象发布必须是线程安全的。例如，在上述例子中我们使用了线程安全的 ArrayBlockingQueue 实例作为传输通道来实现这种对象安全发布。尽管 RecordSet 实例（相当于产品）是在多个线程之间共享的，但是由于生产者线程将 RecordSet 实例发布到消费者线程的对象发布是线程安全的，并且 RecordSet 实例被发布到消费者线程之后仅有一个线程（main 线程）对该实例进行访问（读、写），因此我们无须对访问 RecordSet 的代码使用任何同步控制措施。

通常，生产者和消费者的处理能力是不同的，即生产者生产产品的速率和消费者消费产品的速率是不同的，较为常见的情形是生产者的处理能力比消费者的处理能力大。这种情况下，传输通道所起的作用不仅仅作为生产者和消费者之间传递数据的中介，它在一定程度上还起到一个平衡生产者和消费者处理能力的作用。这是因为生产者每生产一个产品就将其存入传输通道，相对于消费者的处理能力，这个存储操作总是比较快的；而消费者需要一个产品就从传输通道中取出一个，相对于生产者的处理能力，这个操作也总是比较快的。

按照生产者线程数量和消费者线程数量的组合来划分，生产者—消费者模式可以分为如表 5-1 所示的几种。

表 5-1 生产者—消费者模式的分类

类 别	生产者线程数量	消费者线程数量
单生产者—单消费者	1	1
单生产者—多消费者	1	$N (N \geq 2)$
多生产者—多消费者	$N (N \geq 2)$	$N (N \geq 2)$
多生产者—单消费者	$N (N \geq 2)$	1

5.5.1 阻塞队列

传输通道相当于如清单 5-10 所示的接口。在该接口中，类型参数 **P** 代表产品的类型，**take** 方法用于从传输通道中取出一个产品，**put** 方法用于往传输通道中存入一个产品。显然，当传输通道为空的时候消费者无法取出一个产品，此时消费者线程可以进行等待，直到传输通道非空，即生产者线程生产了新的产品。当传输通道存储空间满的时候生产者无法往其中存入新的产品，此时生产者线程可以进行等待，直到传输通道非满，即有消费者消费了产品而腾出新的存储空间。生产者线程往传输通道中成功存入产品后就会唤醒等待传输通道非空的消费者线程，而消费者线程从传输通道中取出一个产品之后就会唤醒等待传输通道非满的生产者线程。我们称这种传输通道的运作方式为阻塞式（Blocking），即从传输通道中存入一个产品或者取出一个产品时，相应的线程可能因为传输通道中没有产品或者其存储空间已满而被阻塞（暂停）。

术语 定义

一般而言，一个方法或者操作如果能够导致其执行线程被暂停（生命周期状态为 WAITING 或者 BLOCKED），那么我们就称相应的方法/操作为阻塞方法（Blocking Method）或者阻塞操作。可见，阻塞方法/操作能够导致上下文切换。常见的阻塞方法/操作包括 `InputStream.read()`、`ReentrantLock.lock()`、申请内部锁等。相反，如果一个方法或者操作并不会导致其执行线程被暂停，那么相应的方法/操作就被称为非阻塞方法（Non-blocking Method）或者非阻塞操作。

清单 5-10 对传输通道的抽象

```
/**
 * 对传输通道的抽象
 *
 * @author Viscent Huang
 */
public interface Channel<P> {
    /**
     * 往传输通道中存入一个产品
     *
     * @param product
     *        产品
     */
    void put(P product) throws InterruptedException;

    /**
     * 从传输通道中取出一个产品
     *
     * @return 产品
     */
    P take() throws InterruptedException;
}
```

JDK 1.5 中引入的接口 `java.util.concurrent.BlockingQueue` 定义了一种线程安全的队列——阻塞队列。该接口相当于上述接口的超集。因此，我们也可以直接使用 `BlockingQueue` 的实现类作为传输通道。例如，在上述实战案例中我们就是使用 `BlockingQueue` 的实现类 `ArrayBlockingQueue` 来充当传输通道（代码参见清单 4-9）。`BlockingQueue` 的常用实现类包括 `ArrayBlockingQueue`、`LinkedBlockingQueue` 和 `SynchronousQueue` 等。

术语 定义

阻塞队列按照其存储空间的容量是否受限制来划分，可分为有界队列（`Bounded Queue`）和无界队列（`Unbounded Queue`）。有界队列的存储容量限制是由应用程序指定的，无界队列的最大存储容量为 `Integer.MAX_VALUE`（ $2^{31}-1$ ）个元素。

往队列中存入一个元素（对象）的操作被称为 `put` 操作，从队列中取出一个元素（对象）的操作被称为 `take` 操作。

`put` 操作相当于生产者线程将对象（产品）安全发布到消费者线程。生产者线程执行 `put` 操作前所执行的任何内存操作，对后续执行 `take` 操作的消费者线程而言是可见的、有序的。

当消费者的处理能力低于生产者的处理能力时，产品的生产速率大于消费速率，这会导致队列中的产品积压，即队列中存储的产品会越来越多。由此导致队列中的这些对象（产品）所占用的内存空间以及其他资源越来越多。因此，我们可能需要限制传输通道的存储容量。此时，我们可以使用有界阻塞队列作为传输通道。

使用有界队列作为传输通道的另外一个好处是可以造成“反压”的效果：当消费者的处理能力跟不上生产者的处理能力时，队列中的产品会逐渐积压到队列满。此时生产者会被暂停，直到消费者消费了部分产品而使队列非满，这相当于生产者暂停其产品生产而给消费者一个跟上其步伐的机会。当然，这里的代价是可能增加的上下文切换。

有界队列可以使用 `ArrayBlockingQueue` 或者 `LinkedBlockingQueue` 来实现。`ArrayBlockingQueue` 内部使用一个数组作为其存储空间，而数组的存储空间是预先分配的，因此 `ArrayBlockingQueue` 的 `put` 操作、`take` 操作本身并不会增加垃圾回收的负担。`ArrayBlockingQueue` 的缺点是其内部在实现 `put`、`take` 操作的时候使用的是同一个锁（显式锁），从而可能导致锁的高争用，进而导致较多的上下文切换。

`LinkedBlockingQueue` 既能实现无界队列，也能实现有界队列。`LinkedBlockingQueue` 的其中一个构造器允许我们创建队列的时候指定队列容量。`LinkedBlockingQueue` 的优点是其内部在实现 `put`、`take` 操作的时候分别使用了两个显式锁（`putLock` 和 `takeLock`），这

降低了锁争用的可能性。`LinkedBlockingQueue` 的内部存储空间是一个链表，而链表节点（对象）所需的存储空间是动态分配的，`put` 操作、`take` 操作都会导致链表节点的动态创建和移除，因此 `LinkedBlockingQueue` 的缺点是它可能增加垃圾回收的负担。另外，由于 `LinkedBlockingQueue` 的 `put`、`take` 操作使用的是两个锁，因此 `LinkedBlockingQueue` 维护其队列的当前长度（`size`）时无法使用一个普通的 `int` 型变量而是使用了一个原子变量（`AtomicInteger`）。这个原子变量可能会被生产者线程和消费者线程争用，因此它可能导致额外的开销。

`SynchronousQueue` 可以被看作一种特殊的有界队列。`SynchronousQueue` 内部并不维护用于存储队列元素的存储空间。设 `synchronousQueue` 为一个任意的 `SynchronousQueue` 实例，生产者线程执行 `synchronousQueue.put(E)` 时如果没有消费者线程执行 `synchronousQueue.take()`，那么该生产者线程会被暂停，直到有消费者线程执行了 `synchronousQueue.take()`；类似地，消费者线程执行 `synchronousQueue.take()` 时如果没有生产者线程执行了 `synchronousQueue.put(E)`，那么该消费者线程会被暂停，直到有生产者线程执行了 `synchronousQueue.put(E)`。因此，在使用 `SynchronousQueue` 作为传输通道的生产者—消费者模式中，生产者线程生产好一个产品之后，会等待消费者线程来取走这个产品才继续生产下一个产品，而不像使用 `ArrayBlockingQueue`、`LinkedBlockingQueue` 作为传输通道的情况下生产者线程将生产好的产品存入队列就继续生产下一个产品。从这点来看，`ArrayBlockingQueue`、`LinkedBlockingQueue` 所实现的传输通道更像是一个信箱，邮递员只需要将普通邮件投入指定的邮箱即可，而不必关心收件人何时会取走邮件；而 `SynchronousQueue` 所实现的通道更像是邮递员投送挂号信时与收件人接触的情形——邮递员必须在收件人本人签收后才能够离开。因此，`SynchronousQueue` 适合于在消费者处理能力与生产者处理能力相差不大的情况下使用。否则，由于生产者线程执行 `put` 操作时没有消费者线程执行 `take` 操作，或者消费者线程执行 `take` 操作的时候没有生产者线程执行 `put` 操作的概率比较大，从而可能导致较多的等待（这意味着上下文切换）。

队列可以被看作生产者线程和消费者之间的共享资源，因此资源调度的公平性在队列上也有所体现。占用队列的线程可以对队列进行 `put` 或者 `take` 操作，那么对队列（作为一种资源）的调度就是决定哪个线程可以进行 `put` 或者 `take` 操作的过程。`ArrayBlockingQueue` 和 `SynchronousQueue` 都既支持非公平调度也支持公平调度，而 `LinkedBlockingQueue` 仅支持非公平调度。

如果生产者线程和消费者线程之间的并发程度比较大，那么这些线程对传输通道内部所使用的锁的争用可能性也随之增加。这时，有界队列的实现适合选用 `LinkedBlockingQueue`，否则我们可以考虑 `ArrayBlockingQueue`。

阻塞队列也支持非阻塞式操作（即不会导致执行线程被暂停）。比如，`BlockingQueue` 接口定义的 `offer(E)` 和 `poll()` 分别相当于 `put(E)` 和 `take()` 的非阻塞版。非阻塞式方法通常用特殊的返回值表示操作结果：`offer(E)` 的返回值 `false` 表示入队列失败（队列已满），`poll()` 返回 `null` 表示队列为空。

提示

`LinkedBlockingQueue` 适合在生产者线程和消费者线程之间的并发程度比较大的情况下使用。

`ArrayBlockingQueue` 适合在生产者线程和消费者线程之间的并发程度较低的情况下使用。

`SynchronousQueue` 适合在消费者处理能力与生产者处理能力相差不大的情况下使用。

5.5.2 限购：流量控制与信号量（Semaphore）

使用无界队列作为传输通道的一个好处是 `put` 操作并不会导致生产者线程被阻塞。因此，无界队列的使用不会影响生产者线程的步伐。但是在队列积压的情况下，无界队列中存储的元素可能越来越多，最终导致这些元素所占用的资源过多。因此，一般我们在使用无界队列作为传输通道的时候会同时限制生产者的生产速率，即进行流量控制以避免传输通道中积压过多的产品。这就好比为防止游客“挤爆”景区而在其入口处对游客进行限流。

JDK 1.5 中引入的标准库类 `java.util.concurrent.Semaphore` 可以用来实现流量控制。为了便于讨论，我们把代码所访问的特定资源或者执行特定操作的机会统一看作一种资源，这种资源被称为虚拟资源（Virtual Resource）。`Semaphore` 相当于虚拟资源配额管理器，它可以用来控制同一时间内对虚拟资源的访问次数。为了对虚拟资源的访问进行流量控制，我们必须使相应代码只有在获得相应配额的情况下才能够访问这些资源。为此，相应代码在访问虚拟资源前必须先申请相应的配额，并在资源访问结束后返还相应的配额。`Semaphore.acquire()/release()` 分别用于申请配额和返还配额。`Semaphore.acquire()` 在成功获得一个配额后会立即返回。如果当前的可用配额不足，那么 `Semaphore.acquire()` 会使其执行线程暂停。`Semaphore` 内部会维护一个等待队列用于存储这些被暂停的线程。`Semaphore.acquire()` 在其返回之前总是会将当前的可用配额减少 1。`Semaphore.release()` 会使当前可用配额增加 1，并唤醒相应 `Semaphore` 实例的等待队列中的一个任意等待线程。

清单 5-11 展示了一个基于 `Semaphore` 和 `BlockingQueue` 实现的带流量控制功能的传输通道。

清单 5-11 带流量控制功能的传输通道

```

/**
 * 基于 Semaphore 的支持流量控制的传输通道实现
 *
 * @author Viscent Huang
 *
 * @param <P>
 *         “产品” 类型
 */
public class SemaphoreBasedChannel<P> implements Channel<P> {
    private final BlockingQueue<P> queue;
    private final Semaphore semaphore;

    /**
     * @param queue
     *         阻塞队列，通常是一个无界阻塞队列
     * @param flowLimit
     *         流量限制数
     */
    public SemaphoreBasedChannel(BlockingQueue<P> queue, int flowLimit) {
        this(queue, flowLimit, false);
    }

    public SemaphoreBasedChannel(BlockingQueue<P> queue, int flowLimit,
        boolean isFair) {
        this.queue = queue;
        this.semaphore = new Semaphore(flowLimit, isFair);
    }

    @Override
    public P take() throws InterruptedException {
        return queue.take();
    }

    @Override
    public void put(P product) throws InterruptedException {
        semaphore.acquire(); // 申请一个配额
        try {
            queue.put(product); // 访问虚拟资源
        } finally {
            semaphore.release(); // 返还一个配额
        }
    }
}

```

从清单 5-11 中可以看出，Semaphore 的使用需要注意以下几点。

- Semaphore.acquire()和 Semaphore.release()总是配对使用。应用代码在访问虚拟资源前调用 Semaphore.acquire()来申请配额，并在虚拟资源访问结束后调用 Semaphore.release()来返回配额。由于 Semaphore 本身并不强制这种配对，即一个线程可以在未执行 Semaphore.acquire()的情况下执行 Semaphore.release()，因此 Semaphore.acquire()/release()的配对使用需要由应用代码来保证。这点和锁的获得与释放有所不同，因为一个线程只有在持有某个锁的情况下才能够释放该锁。
- Semaphore.release()调用总是应该放在一个 finally 块中，以避免虚拟资源访问出现异常的情况下当前线程所获得的配额无法返还（类似于锁泄漏）。

注意

- Semaphore.acquire()和 Semaphore.release()总是配对使用的，这点需要由应用代码自身来保证。
 - Semaphore.release()调用总是应该放在一个 finally 块中，以避免虚拟资源访问出现异常的情况下当前线程所获得的配额无法返还。
- 创建 Semaphore 实例时如果构造器中的参数 permits 值为 1，那么所创建的 Semaphore 实例相当于一个互斥锁。与其他互斥锁不同的是，由于一个线程可以在未执行过 Semaphore.acquire()的情况下执行相应的 Semaphore.release()，因此这种互斥锁允许一个线程释放另外一个线程锁所持有的锁。
 - 配额本身可被看作程序执行特定操作前所需持有的资源，因此对配额的调度也涉及公平性问题。默认情况下，Semaphore 采用的是非公平性调度策略，因此在可用配额数为 0 的情况下，一个线程返回一个配额之后获得配额的那个线程可能是等待队列中那个被唤醒的线程，也可能是其他申请配额的活跃线程。

*5.5.3 管道：线程间的直接输出与输入

Java 标准库类 PipedOutputStream 和 PipedInputStream 是生产者—消费者模式的一个具体例子。PipedOutputStream 和 PipedInputStream 分别是 OutputStream 和 InputStream 的一个子类，它们可用来实现线程间的直接输出和输入。所谓“直接”是指从应用代码的角度来看，一个线程的输出可作为另外一个线程的输入，而不必借用文件、数据库、网络连接等其他数据交换中介。

PipedOutputStream 相当于生产者，其生产的产品是字节形式的数据；PipedInputStream 相当于消费者。PipedInputStream 内部使用 byte 型数组维护了一个循环缓冲区（Circular Buffer），这个缓冲区相当于传输通道。在使用 PipedOutputStream、PipedInputStream 进行输出、输入操作前，PipedOutputStream 实例和 PipedInputStream 实例需要建立起关联（Connect）。建立关联的 PipedOutputStream 实例和 PipedInputStream 实例就像一条输送水

流的管道，管道的一端连着注水口（`PipedOutputStream`），另一端连着出水口（`PipedInputStream`）。这样，生产者所生产的数据（相当于水流）通过向 `PipedOutputStream` 实例输出（相当于向管道注水），就可以被消费者通过关联的 `PipedInputStream` 实例所读取（相当于从出水口接水）。`PipedOutputStream` 实例和 `PipedInputStream` 实例之间的关联可以通过调用各自实例的 `connect` 方法实现，也可以通过在创建相应实例的时候将对方的实例指定为自己的构造器参数来实现。

清单 5-12 展示了一个使用 `PipedOutputStream` 和 `PipedInputStream` 实现的从网络上边下载边解析的 RSS（Rich Site Summary）阅读器。

清单 5-12 边下载边解析的 RSS 阅读器

```
public class ConcurrentRSSReader {
    public static void main(String[] args) throws Exception {
        final int argc = args.length;
        String url = argc > 0 ? args[0] : "http://lorem-rss.herokuapp.com/feed";

        // 从网络加载 RSS 数据
        InputStream in = loadRSS(url);
        // 从输入流中解析 XML 数据
        Document document = parseXML(in);

        // 读取 XML 中的数据
        Element eleRss = (Element) document.getFirstChild();
        Element eleChannel = (Element) eleRss.getElementsByTagName("channel").item(
            0);
        // 完整代码见本书配套下载资源
    }

    private static Document parseXML(InputStream in)
        throws ParserConfigurationException, SAXException, IOException {
        // 完整代码见本书配套下载资源
    }

    private static InputStream loadRSS(final String url) throws IOException {
        final PipedInputStream in = new PipedInputStream();
        // 以 in 为参数创建 PipedOutputStream 实例
        final PipedOutputStream pos = new PipedOutputStream(in);

        Thread workerThread = new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    doDownload(url, pos);
                } catch (Exception e) {

```

```

        // RSS 数据下载过程中出现异常时，关闭相关输出流和输入流
        // 注意，此处我们不能像平常那样在 finally 块中关闭相关输出流
        Tools.silentClose(pos, in);
        e.printStackTrace();
    }
} // run 方法结束
}, "rss-loader");

workerThread.start();
return in;
}
static BufferedInputStream issueRequest(String url) throws Exception {
    // 完整代码见本书配套下载资源
}
static void doDownload(String url, OutputStream os) throws Exception {
    ReadableByteChannel readChannel = null;
    WritableByteChannel writeChannel = null;
    try {
        // 对指定的 URL 发起 HTTP 请求
        BufferedInputStream in = issueRequest(url);
        readChannel = Channels.newChannel(in);
        ByteBuffer buf = ByteBuffer.allocate(1024);
        writeChannel = Channels.newChannel(os);
        while (readChannel.read(buf) > 0) {
            buf.flip();
            writeChannel.write(buf);
            buf.clear();
        }
    } finally {
        Tools.silentClose(readChannel, writeChannel);
    }
} // doDownload 结束
}

```

在 `loadRSS` 方法中，我们创建了一个工作者线程 `workerThread` 专门负责下载指定的 RSS 文件。`loadRSS` 方法会创建并返回一个 `PipedInputStream` 实例 `in`。`workerThread.run()` 会创建一个与 `in` 关联的 `PipedOutputStream` 实例 `pos`，并将该实例作为参数传递给 `doDownload(String, OutputStream)`。`doDownload(String, OutputStream)` 会根据其参数中指定的输出流（即上述的 `PipedOutputStream` 实例 `pos`）创建一个 `WritableByteChannel` 实例 `writeChannel`，并将下载的 RSS 数据写入 `writeChannel`。这实际上实现了将下载的 RSS 数据写入 `pos`。`main` 线程则会从输入流 `in` 中读取数据并进行 XML 解析，而 `in` 中的数据来自 `workerThread` 的输出，这就实现了一个线程（`workerThread`）的输出直接作为另外一个线程（`main`）的输入。从并发的角度来看，这实际上是实现了 RSS 数据的边下载（`workerThread` 负责下载）和边解析（`main` 线程负责解析）。

使用 `PipedOutputStream` 和 `PipedInputStream` 时需要注意以下几点。

- `PipedOutputStream` 和 `PipedInputStream` 适合在两个线程间使用，即适用于单生产者—单消费者的情形。在 `PipedOutputStream` 和 `PipedInputStream` 所实现的生产者—消费者模式中，产品不是一个普通的对象而是字节形式的原始数据，因此在生产者线程不止一个或者消费者线程不止一个的情况下，我们往往需要保证产品序列（字节流）的顺序性，而这可能增加代码的复杂性和额外开销，比如为保证数据的顺序性而引入额外的锁所导致的开销。另外，`PipedOutputStream` 和 `PipedInputStream` 不宜在单线程程序中使用，因为那样可能导致无限制的等待（死锁）。
- 输出异常的处理。如果生产者线程在其执行过程中出现了不可恢复的异常，那么消费者线程就会永远也无法读取到新的数据。但是，由于消费者线程和生产者线程不是同一个线程，因此生产者线程中出现了异常，消费者线程是无法直接侦测的，即无法像单线程程序那样通过 `try-catch` 捕获异常。所以，生产者线程出现异常时需要通过某种方式“知会”相应的消费者线程，否则消费者线程可能会无限制地等待新的数据。生产者线程通常可以通过关闭 `PipedOutputStream` 实例来实现这种“知会”。例如在上述例子中，生产者线程 `workerThread` 在 `catch` 块中提前关闭 `PipedOutputStream` 实例 `pos`，以“知会”生产者线程（`main` 线程）其无法继续提供新的数据。

注意

`PipedOutputStream` 和 `PipedInputStream` 适合在单生产者—单消费者模式中使用。

生产者线程发生异常而导致其无法继续提供新的数据时，生产者线程必须主动提前关闭相应的 `PipedOutputStream` 实例（调用 `PipedOutputStream.close()`）。

5.5.4 一手交钱，一手交货：双缓冲与 Exchanger

缓冲（`Buffering`）是一种常用的数据传递技术。缓冲区相当于数据源（`Source`，即数据的原始提供方）与数据使用方（`Sink`）之间的数据容器。从这个角度来看，数据源相当于生产者，数据使用方相当于消费者。数据源所提供的数据相当于产品，而缓冲区可被看作产品的容器或者外包装。在多线程环境下，有时候我们会使用两个（或者更多）缓冲区来实现数据从数据源到数据使用方的移动。其中一个缓冲区填充满来自数据源的数据后可以被数据使用方进行“消费”，而另外一个空的（或者已经使用过的）缓冲区则用来填充来自数据源的新的数据。这里，负责填充缓冲区的是一个线程（生产者线程），而使用已填充完毕的另外一个缓冲区的则是另外一个线程（消费者线程）。因此，当消费者线程消费一个已填充的缓冲区时，另外一个缓冲区可以由生产者线程进行填充，从而实现了数据生成与消费的并发。这种缓冲技术就被称为双缓冲（`Double Buffering`）。

JDK 1.5 中引入的标准库类 `java.util.concurrent.Exchanger` 可以用来实现双缓冲。`Exchanger` 相当于一个只有两个参与方的 `CyclicBarrier`。`Exchanger.exchange(V)` 相当于 `CyclicBarrier.await()`。`Exchanger.exchange(V)` 的声明如下：

```
public V exchange(V x) throws InterruptedException
```

其中，`V` 是 `Exchanger` 类的类型参数，参数 `x` 和返回值相当于缓冲区。

通常，初始状态下生产者和消费者各自创建一个空的缓冲区。消费者线程执行 `Exchanger.exchange(V)` 时将参数 `x` 指定为一个空的或者已经使用过的缓冲区，生产者线程执行 `Exchanger.exchange(V)` 时将参数 `x` 指定为一个已经填充完毕的缓冲区。比照 `CyclicBarrier` 来说，生产者线程和消费者线程都执行到 `Exchanger.exchange(V)` 相当于这两个线程都到达了集合点，此时生产者线程和消费者线程各自对 `Exchanger.exchange(V)` 的调用就会返回。`Exchanger.exchange(V)` 的返回值是对方线程执行该方法时所指定的参数 `x` 的值。因此，`Exchanger.exchange(V)` 的返回就造成一种生产者线程和消费者线程之间交换缓冲区（产品）的效果，即消费者线程向生产者线程提供（通过指定参数 `x` 的值）的是一个空的（或者已经使用过的）的缓冲区，而生产者线程向消费者线程提供（通过指定参数 `x` 的值）的则是一个已经填充完毕的缓冲区。这就好比当面交易的情况下，交易双方“一手交钱，一手交货”。这样，生产者线程和消费者线程之间通过不断地交换缓冲区（相当于产品的容器）就实现了将生产者所生产的一个个产品传递给消费者的效果。因此，`Exchanger` 从逻辑上可以被看作一种 `SynchronousQueue`，其内部也不维护用于存储产品的存储空间。

在单生产者—单消费者模式中，我们可以考虑使用 `Exchanger` 作为传输通道。例如，在第 4 章的第 2 个实战案例（响应延时统计程序，代码见清单 4-7）中，我们也可以使用 `Exchanger` 作为生产者线程（日志文件读取线程）和消费者线程（统计处理线程）之间的传输通道。为此，我们只需要新建 `AbstractLogReader` 的子类来实现日志读取线程即可，如清单 5-13 所示。

清单 5-13 使用 `Exchanger` 作为传输通道实例

```
public class ExchangerBasedLogReaderThread extends AbstractLogReader {
    private final Exchanger<RecordSet> exchanger;
    private volatile RecordSet nextToFill;
    private RecordSet consumedBatch;

    public ExchangerBasedLogReaderThread(InputStream in, int inputBufferSize,
        int batchSize) {
        super(in, inputBufferSize, batchSize);
        exchanger = new Exchanger<RecordSet>();
    }
}
```

```

    nextToFill = new RecordSet(batchSize);
    consumedBatch = new RecordSet(batchSize);
}

@Override
protected RecordSet getNextToFill() {
    return nextToFill;
}

@Override
protected void publish(RecordSet recordSet) throws InterruptedException {
    nextToFill = exchanger.exchange(recordSet);
}

@Override
protected RecordSet nextBatch() throws InterruptedException {
    consumedBatch = exchanger.exchange(consumedBatch);
    if (consumedBatch.isEmpty()) {
        consumedBatch = null;
    }
    return consumedBatch;
}
}

```

从双缓冲的角度来看，ExchangerBasedLogReaderThread 内部维护了两个类型为 RecordSet（参见清单 4-8）的缓冲区 nextToFill 和 consumedBatch，前者表示待填充的缓冲区，后者表示填充后已被“消费”过的缓冲区。ExchangerBasedLogReaderThread.nextBatch() 的执行线程（统计处理线程，参见清单 4-7）相当于消费者线程，它提供一个已“消费”过的缓冲区来调用 exchanger.exchange(consumedBatch) 以获得（交换到）一个新填充的缓冲区。ExchangerBasedLogReaderThread.publish(RecordSet) 的执行线程（日志文件读取线程，即 ExchangerBasedLogReaderThread 实例）相当于生产者线程，它提供一个新填充完毕的缓冲区来调用 exchanger.exchange(recordSet) 以获得（交换到）一个待填充的缓冲区。

5.5.5 一个还是一批：产品的粒度

在第 4 章的第 2 个实战案例（响应延时统计程序，代码见清单 4-7）所实现的生产者—消费者模式中，RecordSet 类相当于产品，而一个 RecordSet 实例可以包含一批日志记录（例如 2000 条记录）。因此，该实例中生产者线程（文件读取线程）和消费者线程（统计处理线程）之间传递的产品是一批记录而不是一条记录！显然，如果该实例以一条日志记录代表一个产品的话，那么由于待统计的日志记录可达上千万条之多，因此产品在传输通道上的移动操作（put 和 take 操作）次数将可能达到千万级。而将一批日志记录作为一

一个产品则可以大幅减少产品在传输通道上的移动次数，从而可减少相应的开销。这里，一条日志记录可以作为一个产品，而一批日志记录也可以作为一个产品。这就是产品的粒度（Granularity）。在上述例子中，使用一条日志记录来表示的产品粒度过细，使用一批日志记录来表示的产品粒度较粗。

产品的粒度是一个相对的概念。在问题规模一定的情况下，产品的粒度过细会导致产品在传输通道上的移动次数增大；产品的粒度稍微大些可以减少产品在传输通道上的移动次数，但是产品所占用的资源也随之增加。因此，产品粒度的确定是权衡产品在传输通道上的移动次数和产品所占用的资源的结果。

5.5.6 再探线程与任务之间的关系

在生产者—消费者模式中，一个产品也可以代表消费者线程需要执行的任务。即使是在单生产者—单消费者模式中一个生产者线程也可以生产多个产品（任务），而这些产品所代表的任务都是由一个消费者线程负责执行（消费）的。因此，线程和任务之间可以是一对多的关系，即一个线程可以先后执行多个任务。从这点来看，生产者—消费者模式有利于充分利用有限的线程资源：一个线程可以执行多个而不是一个任务。例如，清单 5-14 展示了一个通用的任务执行器 TaskRunner。TaskRunner 的实例变量 channel 相当于传输通道，TaskRunner 内部维护的工作者线程相当于消费者线程。TaskRunner.submit(Runnable) 的执行线程相当于生产者线程。生产者只需要调用 TaskRunner.submit(Runnable) 提交一个任务（Runnable 接口实例，相当于产品），该任务即可以被 TaskRunner 执行。显然，一个 TaskRunner 实例（对应一个工作者线程）可以用于执行生产者提交的多个任务。

清单 5-14 通用任务执行器

```
public class TaskRunner {
    protected final BlockingQueue<Runnable> channel;
    protected volatile Thread workerThread;
    public TaskRunner(BlockingQueue<Runnable> channel) {
        this.channel = channel;
        this.workerThread = new WorkerThread();
    }

    public TaskRunner() {
        this(new LinkedBlockingQueue<Runnable>());
    }

    public void init() {
        final Thread t = workerThread;
        if (null != t) {
            t.start();
        }
    }
}
```

```

    }
}

public void submit(Runnable task) throws InterruptedException {
    channel.put(task);
}

class WorkerThread extends Thread {
    @Override
    public void run() {
        Runnable task = null;
        try {
            // 注意：下面这种代码写法实际上可能导致工作者线程永远无法终止！
            // 在 5.6 节中我们将会解决这个问题。
            for (;;) {
                task = channel.take();
                try {
                    task.run();
                } catch (Throwable e) {
                    e.printStackTrace();
                }
            }
            /// for 循环结束
        } catch (InterruptedException e) {
            // 什么也不做
        }
        /// run 方法结束
    }
}
}

```

5.6 对不起，打扰一下：线程中断机制

线程间协作还有一种常见的形式是，一个线程请求另外一个线程停止其正在执行的操作。比如，对于有些比较耗时的任务，我们往往会采用专门的工作者线程来负责其执行，如果中途要取消（比如用户不想等了）这类任务的执行，那么我们就需要借助 Java 线程中断机制。

Java 线程中断机制相当于 Java 线程与线程间协作的一套协议框架（合同范本）。中断（Interrupt）可被看作由一个线程（发起线程 *Originator*）发送给另外一个线程（目标线程 *Target*）的一种指示（Indication），该指示用于表示发起线程希望目标线程停止其正在执行的操作。中断仅代表发起线程的一个诉求，而这个诉求能否被满足则取决于目标线程自身——目标线程可能会满足发起线程的诉求，也可能根本不理睬发起线程的诉求！Java 平台会为每个线程维护一个被称为中断标记（Interrupt Status）的布尔型状态变量用于表示相应线程是否接收到了中断，中断标记值为 `true` 表示相应线程收到了中断。目标线程可以

通过 `Thread.currentThread().isInterrupted()` 调用来获取该线程的中断标记值，也可以通过 `Thread.interrupted()` 来获取并重置（也称清空）中断标记值，即 `Thread.interrupted()` 会返回当前线程的中断标记值并将当前线程中断标记重置为 `false`。调用一个线程的 `interrupt()` 相当于将该线程（目标线程）的中断标记置为 `true`。

目标线程检查中断标记后所执行的操作，被称为目标线程对中断的响应，简称中断响应。设有个发起线程 `originator` 和目标线程 `target`，那么 `target` 对中断的响应一般包括：

- 无影响。`originator` 调用 `target.interrupt()` 不会对 `target` 的运行产生任何影响。这种情形也可以称为目标线程无法对中断进行响应。`InputStream.read()`、`ReentrantLock.lock()` 以及申请内部锁等阻塞方法/操作就属于这种类型。
- 取消任务的运行。`originator` 调用 `target.interrupt()` 会使 `target` 在侦测到中断（即中断标记值为 `true`）那一刻所执行的任务被取消（中止），而这并不会运行 `target` 继续处理其他任务。
- 工作者线程停止。`originator` 调用 `target.interrupt()` 会使 `target` 终止，即 `target` 的生命周期状态变更为 `TERMINATED`。

InterruptedException 异常处理及中断响应

Java 标准库中的许多阻塞方法对中断的响应方式都是抛出 `InterruptedException` 等异常，如表 5-2 所示。Java 标准库中也有些阻塞方法/操作无法响应中断，例如 `InputStream.read()`、`Lock.lock()` 以及内部锁的申请。

表 5-2 能够对中断做出响应的一些标准库类/方法

方法（或者类）	为响应中断而抛出的异常
<code>Object.wait()/wait(long)/wait(long, int)</code>	<code>InterruptedException</code>
<code>Thread.sleep(long) /sleep(long, int)</code>	<code>InterruptedException</code>
<code>Thread.join()/join(long)/join(long, int)</code>	<code>InterruptedException</code>
<code>java.util.concurrent.BlockingQueue.take() /put(E)</code>	<code>InterruptedException</code>
<code>java.util.concurrent.locks.Lock.lockInterruptibly()</code>	<code>InterruptedException</code>
<code>java.util.concurrent.CountDownLatch.await()</code>	<code>InterruptedException</code>
<code>java.util.concurrent.CyclicBarrier.await()</code>	<code>InterruptedException</code>
<code>java.util.concurrent.Exchanger.exchange(V)</code>	<code>InterruptedException</code>
<code>java.nio.channels.InterruptibleChannel</code>	<code>java.nio.channels.ClosedByInterruptException</code>

能够响应中断的方法通常是在执行阻塞操作前判断中断标志,若中断标志值为 true 则抛出 `InterruptedException`。例如, `ReentrantLock.lockInterruptibly()` 的功能与 `ReentrantLock.lock()` 类似,二者都能用于申请相应的显式锁,但是 `ReentrantLock.lockInterruptibly()` 能够对中断做出响应。`ReentrantLock.lockInterruptibly()` 方法对中断的响应是通过其调用的一个名为 `acquireInterruptibly` 的方法实现的。`acquireInterruptibly` 方法会在执行申请锁这个阻塞操作前检查当前线程的中断标记,若中断标记值为 true 则抛出 `InterruptedException` 异常,如清单 5-15 所示。依照惯例,凡是抛出 `InterruptedException` 异常的方法,通常会在其抛出该异常之前将当前线程的线程中断标记重置为 false。因此, `acquireInterruptibly` 方法在判断中断标记时调用的是 `Thread.interrupted()` 而非 `Thread.currentThread().isInterrupted()`。

清单 5-15 `ReentrantLock.lockInterruptibly()` 对中断的响应

```
public final void acquireInterruptibly(int arg)
    throws InterruptedException {
    if (Thread.interrupted())
        throw new InterruptedException();
    if (!tryAcquire(arg))
        doAcquireInterruptibly(arg);
}
```

注意 依照惯例,抛出 `InterruptedException` 异常的方法,通常会在其抛出该异常时将当前线程的线程中断标记重置为 false。

如果发起线程给目标线程发送中断的那一刻,目标线程已经由于执行了一些阻塞方法/操作而被暂停(生命周期状态为 `WAITING` 或者 `BLOCKED`)了,比如清单 5-15 中的方法已经执行到了第 2 个 if 语句,那么此时 Java 虚拟机可能会设置目标线程的线程中断标记并将该线程唤醒,从而使目标线程被唤醒后继续执行的代码再次得到响应中断的机会。因此,这种情形下能够响应中断的阻塞方法/操作依然可以抛出 `InterruptedException`,并在此之前将线程中断标记清空。例如,目标线程可能因为执行 `CountDownLatch.await()`、`CyclicBarrier.await()` 以及 `ReentrantLock.lockInterruptibly()` 等能够响应中断的阻塞方法/操作而被暂停时,发起线程给这些方法的执行线程发送中断会导致 Java 虚拟机将相应的线程唤醒并使其抛出 `InterruptedException`。可见,给目标线程发送中断还能够产生唤醒目标线程的效果。

因此,Java 应用层代码通常可以通过对 `InterruptedException` 等异常进行处理的方式来实现中断响应。对 `InterruptedException` 异常的正确处理方式包括以下几种。

- 不捕获 `InterruptedException`。如果应用代码的某个方法调用了能够对中断进行响应的阻塞方法,那么我们可以选择在这个方法的异常声明(throws)中也加一

个 `InterruptedException`。这种做法实质上是当前方法不知道如何处理中断比较恰当，因此将“难题”抛给其上层代码（比如这个方法的调用方）。

- 捕获 `InterruptedException` 后重新将该异常抛出。使用这种策略通常是由于应用代码需要捕获 `InterruptedException` 并对此做一些中间处理（比如处理部分完成的任务），接着再将“难题”抛给其上层代码。
- 捕获 `InterruptedException` 并在捕获该异常后中断当前线程。这种策略实际上在捕获到 `InterruptedException` 后又恢复中断标志，这相当于当前代码告诉其他代码：“我发现了中断，但我并不知道如何处理比较妥当，因此我为你保留了中断标记，你看着办吧！”本书源码所用的工具类 `Tools` 的 `randomPause` 方法就采用了这种处理策略，如清单 5-16 所示。

清单 5-16 捕获 `InterruptedException` 后恢复中断标志

```
public final class Tools {
    public static void randomPause(int maxPauseTime) {
        int sleepTime = rnd.nextInt(maxPauseTime);
        try {
            Thread.sleep(sleepTime);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt(); // 保留线程中断标记
        }
    }
    // ...
}
```

比较危险的一种处理方法是“吞没”（Swallow）`InterruptedException`，即应用代码在捕获 `InterruptedException` 之后既不重新抛出也不保留中断标志。本书在本章之前的所有源码中只要涉及 `InterruptedException` 的都采用这种处理策略，这其实只是为了避免讲解上的不便。事实上，这种处理策略只有在线程捕获到 `InterruptedException` 就可以终止的情况下才适用，其他情况下使用该策略可能导致目标线程无法被终止。

应用代码也可以先判断中断标记，若中断标记值为 `true`，则直接执行相应的中断响应操作。

5.7 线程停止：看似简单，实则不然

某些情况下，我们可能需要主动停止线程而不是等待线程自然终止（`run` 方法返回）。一些典型场景如下。

- 服务或者系统关闭。当一个服务不再被需要的时候，我们应该及时停止该服务所

启动的工作者线程以节约宝贵的线程资源。由于非守护线程（用户线程）会阻止 Java 虚拟机正常关闭，因此在系统停止前所有用户线程都应该先行停止。

- 错误处理。同质（线程的任务处理逻辑相同）工作者线程中的一个线程出现不可恢复的异常时，其他线程往往就没有必要继续运行下去了，此时我们需要主动停止其他工作者线程。例如，第4章的第1个实战案例（大文件下载）中的一个下载线程如果出现了不可恢复的异常，那么其他下载线程即使运行到自然终止，最终整个大文件下载也还是失败的（文件不完整），这时我们就需要将其他下载线程主动停止掉。
- 用户取消任务。在某些比较耗时的任务执行过程中用户可能会取消这个任务，这时任务的取消往往是通过主动停止相应的工作者线程实现的。

然而，停止线程却是目标简单但实现并不那么简单的一件事情：首先，Java 标准库并没有提供可以直接停止线程的 API⁴；其次，停止线程的时候有一些额外的细节需要考虑（下文会介绍）。

我们不难想到主动停止一个线程的实现思路：为待停止的线程（目标线程）设置一个线程停止标记（布尔型数据），目标线程检测到该标志值为 `true` 时则设法让其 `run` 方法返回，这样就实现了线程的终止。依照这个思路，乍一看似乎线程中断标记可以作为线程停止标记，而目标线程则可以通过响应中断来实现其停止，但是由于线程中断标记可能会被目标线程所执行的某些方法清空，因此从通用性的角度来看线程中断标记并不能作为线程停止标记！例如，上文的通用任务执行器 `TaskRunner`（参见清单 5-14）中维护的工作者 `workerThread` 看起来似乎是可以借助 `workerThread.interrupt()` 调用来停止的——因为 `workerThread.run()` 对 `channel.take()`（`BlockingQueue.take()`）的调用可能由于其他线程调用 `workerThread.interrupt()` 而抛出 `InterruptedException`（响应中断），并且 `workerThread` 对该异常的处理方式是捕获并在捕获后使其 `run` 方法返回，如下代码片段所示。

```
public void run() {
    Runnable task = null;
    // 注意：下面这种代码写法实际上可能导致工作者线程永远无法终止！
    try {
        for (;;) {
            task = channel.take();
            try {
                task.run();
            } catch (Throwable e) {
                e.printStackTrace();
            }
        }
    }
```

4 `Thread.stop()`早已是被废弃的方法了。

```

    }
    } // for 循环结束
} catch (InterruptedException e) {
    // 什么也不做
}
} // run 方法结束

```

实际上，由于发起线程在执行 `workerThread.interrupt()` 的时候 `workerThread` 可能正在执行 `task.run()`，而 `task.run()` 中的代码可能会清除（“吞没”）线程中断标记，从而使得 `workerThread` 依旧无法终止，如清单 5-17 所示。

清单 5-17 线程中断标记不能作为线程停止标记 Demo

```

public class MayNotBeTerminatedDemo {
    public static void main(String[] args) throws InterruptedException {
        TaskRunner tr = new TaskRunner();
        tr.init();
        tr.submit(new Runnable() {
            @Override
            public void run() {
                Debug.info("before doing task");
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    // 什么也不做:这会导致线程中断标记被清除
                }
                Debug.info("after doing task");
            }
        });
        tr.workerThread.interrupt();
    }
}

```

运行上述程序，我们可以看到类似如下的输出：

```

[2016-08-21 21:42:46.811][INFO][Thread-0]:before doing task
[2016-08-21 21:42:46.812][INFO][Thread-0]:after doing task

```

但是，`workerThread` 却依然未终止。由此可见，从通用的角度来看，我们不能使用线程中断标记作为线程停止标记，而需要使用一个专门的实例变量来作为线程停止标记。

但是，光使用专门的实例变量来作为线程停止标记仍然不够，这是由于当线程停止标记置为 `true`（表示目标线程需要被停止）的时候，目标线程可能因为执行了一些阻塞方法（比如 `CountDownLatch.await()`）而被暂停，因此，这时线程停止标记压根儿不会对目标线程产生任何影响！由此可见，为了使线程停止标记的设置能够起作用，我们可能还需要给

目标线程发送中断以将其唤醒，使之得以判断线程停止标记。

另外，在生产者—消费者模式中一个线程试图停止目标线程的时候，该线程可能仍然有尚未处理完毕的任务，因此我们可能需要以“优雅”的方式将该线程停止——目标线程只有在其处理完所有待处理任务之后才能够终止。

综上所述，一个比较通用且能够以优雅的方式实现线程停止的方案如清单 5-18 所示。

清单 5-18 通用的线程优雅停止办法实例

```
public class TerminatableTaskRunner implements TaskRunnerSpec {
    protected final BlockingQueue<Runnable> channel;
    // 线程停止标记
    protected volatile boolean inUse = true;
    // 待处理任务计数器
    public final AtomicInteger reservations = new AtomicInteger(0);
    private volatile Thread workerThread;
    public TerminatableTaskRunner(BlockingQueue<Runnable> channel) {
        this.channel = channel;
        this.workerThread = new WorkerThread();
    }

    public TerminatableTaskRunner() {
        this(new LinkedBlockingQueue<Runnable>());
    }

    @Override
    public void init() {
        final Thread t = workerThread;
        if (null != t) {
            t.start();
        }
    }

    @Override
    public void submit(Runnable task) throws InterruptedException {
        channel.put(task);
        reservations.incrementAndGet(); // 语句①
    }

    public void shutdown() {
        Debug.info("Shutting down service...");
        inUse = false; // 语句②
        final Thread t = workerThread;
        if (null != t) {
            t.interrupt(); // 语句③
        }
    }
}
```

```

    }

    class WorkerThread extends Thread {
        @Override
        public void run() {
            Runnable task = null;
            try {
                for (;;) {
                    // 线程不再被需要，且无待处理任务
                    if (!inUse && reservations.get() <= 0) { // 语句④
                        break;
                    }
                    task = channel.take();
                    try {
                        task.run();
                    } catch (Throwable e) {
                        e.printStackTrace();
                    }
                    // 使待处理任务数减少 1
                    reservations.decrementAndGet(); // 语句⑤
                } // for 循环结束
            } catch (InterruptedException e) {
                workerThread = null;
            }
            Debug.info("worker thread terminated.");
        } // run 方法结束
    } // WorkerThread 结束
}

```

这里，我们使用布尔型变量 `inUse` 作为线程停止标记，使用原子变量 `reservations` 表示目标线程待处理任务的数量（即传输通道中任务的数量）。`submit` 方法每接收到一个提交的任务时便将 `reservations` 的值增加 1（语句①）。在 `shutdown` 方法中，我们在将 `inUse` 置为 `false`（语句②）的时候还向目标线程发送中断（语句③）。接着，我们使目标线程的 `run` 方法每次从传输通道中取出一个任务前判断线程停止标记和待处理任务的数量（语句④）。若此时客户端不会再提交新的任务（`inUse==false`）且无待处理任务（`reservations.get()≤0`），那么目标线程就可以优雅终止了；否则，目标线程从传输通道中取出一个任务执行后，会将待处理任务数减 1（语句⑤）。目标线程的 `run` 方法还对 `InterruptedException` 进行了捕获，并在捕获到该异常后使其返回（线程随之终止）。这里，`run` 方法所捕获的异常只可能是 `channel.take()` 调用所抛出的。由于我们不仅仅对中断进行了处理，还在每次取出待处理任务前判断了线程停止标记，因此，即使是客户端代码在调用 `shutdown` 方法那一刻，目标线程正在执行 `task.run()` 且 `task.run()` 中的代码清空了线程中断标记，而使得后续执行的 `channel.take()` 调用无法抛出 `InterruptedException`（因为线程中断标记被 `task.run()`）

中的代码清空了，如清单 5-17 所示）的情况下，目标线程也还有退路——它能够通过对线程停止标记的判断而实现停止。

5.7.1 生产者—消费者模式中的线程停止

在生产者—消费者模式中，生产者线程需要先于消费者线程停止，否则生产者所生产的产品会无法被处理。在单生产者—单消费者模式中，停止生产者、消费者线程有一种简单的方法：生产者线程在其终止（可以是自然终止）前往传输通道中存入一个特殊产品作为消费者线程的线程停止标志，消费者线程取出这个产品之后就可以退出 run 方法而终止了。比如，第 4 章的第 2 个实战案例（响应延时统计程序，参见清单 4-7）是一个单生产者—单消费者模式实例，该案例中的日志读取线程（生产者线程）在读取完所有日志记录之后就主动退出 run 方法了，但在此之前它会往传输通道中存入一个空的日志记录集（即未经过填充）。统计处理线程（消费者线程）从传输通道中取出一个空的日志记录集（该记录集最终会被转换为 null）之后也主动退出 run 方法。这样就实现了统计完毕后生产者线程和消费者线程自动终止的效果。这种办法虽然简单，但是生产者线程之间、消费者线程之间的并发使得这种办法应用到多生产者线程或者多消费者线程中会比较困难。此时，我们需要使用清单 5-18 中的方案。

5.7.2 实践：Web 应用中的线程停止

Java Web 应用中应用代码自身所启动的线程，比如在 ServletContextListener.contextInitialized(ServletContextEvent)或者 Servlet.init()中启动的线程，在该 Web 应用停止的时候如果仍在运行的，那么该 Web 应用停止后这些线程（即使是守护线程）也可能仍然在运行。这是因为 Web 应用被停止的时候其所在的 Web 服务器（容器）仍然在运行，即相应的 Java 进程仍然还在，所以该进程中启动的线程如果没有被主动停止，那么它可能还在运行。这些线程（对象）无法被垃圾回收就会导致它们所引用的对象也无法被垃圾回收，从而可能导致内存泄漏！

某些 Web 服务器考虑到了这一点并对此做了一些补救的措施。例如，Tomcat 6.0.37 会在 Web 应用停止的时候检测是否存在由 Web 应用自身启动且未结束的线程，如果有这样的线程，那么 Tomcat 会尽其最大努力来将这些线程停止。尽管如此，Tomcat 还是无法保证这些线程能够完全被停止，即使能够停止也无法保证这种停止是优雅的⁵。因此，我

5 以 Tomcat 6.0.37 为例，Tomcat 最终会调用 Thread.stop()这个被废弃的方法来强行停止这些线程。而 Thread.stop()不一定就能够将目标线程停止，并且它无法以优雅的方式停止线程。

们不能依赖 Web 服务器，而是要在应用停止时自行将这些线程停止。

为此，我们可以维护一个线程终止登记表 ThreadTerminationRegistry，用于记录哪些线程是需要 Web 应用停止时被主动停止的，如清单 5-19 所示。

清单 5-19 线程终止登记表源码

```
/**
 * 线程终止登记表
 *
 * @author Viscent Huang
 */
public enum ThreadTerminationRegistry {
    INSTANCE;
    private final Set<Handler> handlers = new HashSet<Handler>();
    public synchronized void register(Handler handler) {
        handlers.add(handler);
    }

    public void clearThreads() {
        // 为保障线程安全，在遍历时将 handlers 复制一份
        final Set<Handler> handlersSnapshot;
        synchronized (this) {
            handlersSnapshot = new HashSet<Handler>(handlers);
        }

        for (Handler handler : handlersSnapshot) {
            try {
                handler.terminate();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
}

/**
 * 线程终止处理器
 * <p>
 * 封装了有关线程停止的知识
 *
 * @author Viscent Huang
 */
public static interface Handler {
    void terminate();
}
```

应用程序每创建一个（或者多个）不会自动终止的工作者线程（这类线程的 run 方法

体通常是一个循环语句)时,就调用 ThreadTerminationRegistry.register(Handler)来登记一个线程终止处理器(ThreadTerminationRegistry.Handler 实例),如清单 5-20 中的语句③所示。线程终止处理器的 terminate 方法封装了相应线程(一个或者多个)的终止逻辑,如清单 5-20 中的语句②所示。当 Web 应用停止的时候,我们就通过 ThreadTerminationRegistry.clearThreads()调用主动将所有登记过的线程停止,如清单 5-20 中的语句①所示。

清单 5-20 在 Web 应用中实现线程停止

```
public class ThreadManagementContextListener implements ServletContextListener {

    @Override
    public void contextDestroyed(ServletContextEvent ctxEvt) {
        // 停止所有登记的线程
        ThreadTerminationRegistry.INSTANCE.clearThreads(); // 语句①
    }

    @Override
    public void contextInitialized(ServletContextEvent ctxEvt) {
        // 创建并启动一个数据库监控线程
        AbstractMonitorThread databaseMonitorThread;
        databaseMonitorThread = new AbstractMonitorThread(
            2000) {
            @Override
            protected void doMonitor() {
                Debug.info("Monitoring database...");
                // ...

                // 模拟实际的时间消耗
                Tools.randomPause(100);
            }
        };
        databaseMonitorThread.start();
    }

    /**
     * 抽象监控线程
     */
    * @author Viscent Huang
    */
    static abstract class AbstractMonitorThread extends Thread {
        // 监控周期
        private final long interval;
        // 线程停止标记
        final AtomicBoolean terminationToken = new AtomicBoolean(false);

        public AbstractMonitorThread(long interval) {
            this.interval = interval;
        }
    }
}
```

```

// 设置为守护线程!
setDaemon(true);
ThreadTerminationRegistry.Handler handler;
handler = new ThreadTerminationRegistry.Handler() {
    @Override
    public void terminate() {
        terminationToken.set(true);
        AbstractMonitorThread.this.interrupt();
    }
}; // 语句②
ThreadTerminationRegistry.INSTANCE.register(handler); // 语句③
}

@Override
public void run() {
    try {
        while (!terminationToken.get()) {
            doMonitor();
            Thread.sleep(interval);
        }
    } catch (InterruptedException e) {
        // 什么也不做
    }
    Debug.info("terminated:%s", Thread.currentThread());
}

// 子类覆盖该方法来实现监控逻辑
protected abstract void doMonitor();
}
}

```

5.8 本章小结

本章介绍了多线程编程中线程间常见的协作形式以及 Java 平台对这些协作形式所提供的支持，本章知识结构如图 5-6 所示。

等待线程可以通过执行 `Object.wait()/wait(long)` 来实现等待。通知线程可以通过执行 `Object.notify()/notifyAll()` 来实现通知。等待线程、通知线程在执行 `Object.wait()/wait(long)`、`Object.notify()/notifyAll()` 时必须持有相应对象对应的内部锁。为了避免信号丢失问题以及欺骗性唤醒问题，等待线程将等待线程对保护条件的判断、`Object.wait()/wait(long)` 的调用必须放在相应对象所引导的临界区中的一个循环语句之中。

使用 `notify()` 替代 `notifyAll()` 必须使以下两个条件同时得以满足：

- 一次通知仅需要唤醒至多一个线程；

- 相应对象上的所有等待线程都是同质等待线程。

使用 `notify()` 替代 `notifyAll()` 可以减少等待/通知中产生的上下文切换。通知线程在执行完 `Object.notify()/notifyAll()` 后尽快释放相应对象的内部锁也有助于减少上下文切换。

条件变量 (`Condition` 接口) 是 `wait/notify` 的替代品。`Condition` 接口的 API 与 `wait/notify` 类似: `Condition.await()/awaitUntil(Date)` 相当于 `Object.wait()/wait(long)`; `Condition.signal()/signalAll()` 相当于 `Object.notify()/notifyAll()`。`Condition.awaitUntil(Date)` 解决了 `Object.wait(long)` 存在的问题——无法区分其返回是否是由等待超时而导致的。

`Condition` 接口本身只是对解决过早唤醒问题提供了支持。要真正解决过早唤醒问题, 我们需要通过应用代码维护保护条件与条件变量之间的对应关系, 即使用不同保护条件的等待线程需要调用不同的条件变量的 `await` 方法来实现其等待, 并使通知线程在更新了相关共享变量之后, 仅调用与这些共享变量有关的保护条件所对应的条件变量的 `signal/signalAll` 方法来实现通知。

`CountDownLatch` 能够用来实现一个线程等待其他线程执行的特定操作的结束。等待线程执行 `CountDownLatch.await()`, 通知线程执行 `CountDownLatch.countDown()`。为避免等待线程永远处于暂停状态而无法被唤醒, `CountDownLatch.countDown()` 调用通常需要被放在 `finally` 块中。一个 `CountDownLatch` 实例只能实现一次等待/通知。对于同一个 `CountDownLatch` 实例 `latch`, `latch.countDown()` 的执行线程在执行该方法之前所执行的任何内存操作, 对等待线程在 `latch.await()` 调用返回之后的代码是可见的且有序的。

`CyclicBarrier` 能够用于实现多个线程间的相互等待。`CyclicBarrier.await()` 既是等待方法又是通知方法。`CyclicBarrier` 实例的所有参与方除最后一个线程外都相当于等待线程, 最后一个线程则相当于通知线程。与 `CountDownLatch` 不同的是, `CountDownLatch` 实例是可以复用的——一个 `CountDownLatch` 实例可以实现多次等待/通知。在使用 `CountDownLatch` 足以满足要求的情况下, 我们应该避免使用 `CyclicBarrier`。`CyclicBarrier` 的典型应用场景包括: 使迭代 (Iterative) 算法并发化, 在测试代码中模拟高并发。

在生产者—消费者模式中, 生产者负责生产产品并通过传输通道将产品以线程安全的方式发布到消费者线程。消费者线程仅负责从传输通道中取出产品进行“消费”。产品既可以是数据, 也可以是待处理的任务。`BlockingQueue` 的实现类 `ArrayBlockingQueue`、`LinkedBlockingQueue` 和 `SynchronousQueue` 等以及 `Exchanger` 类可作为传输通道。

生产者与消费者所执行的处理, 即产品的生产与“消费”是并发的。这使得我们能够平衡生产者、消费者处理能力的差异, 即避免了一方处理过慢对另一方产生影响。另外,

生产者—消费者模式使得一个线程（消费者线程）可以处理多个任务，提高了线程的利用率。

使用无界队列作为传输通道时往往需要借助 Semaphore 控制生产者的生产速率。Semaphore 相当于能够对程序访问虚拟资源的并发程度进行控制的配额调度器。Semaphore.acquire()用于申请配额，Semaphore.release()用于返还配额，Semaphore.release()调用总是放在 finally 块中。Semaphore.acquire()和 Semaphore.release()总是配对使用的，这点需要由应用代码来确保。Semaphore 对配额的调度既支持非公平策略（默认策略），也支持公平策略。

PipedOutputStream/PipedInputStream 是 Java 标准库类中生产者—消费者模式的一个具体例子。PipedOutputStream/PipedInputStream 适合在单生产者—单消费者模式中使用，应避免在单线程程序中使用 PipedOutputStream/PipedInputStream。生产者线程发生异常而导致其无法继续提供新的数据时，生产者线程必须主动提前关闭相应的 PipedOutputStream 实例（调用 PipedOutputStream.close()）。

Exchanger 类也可作为传输通道，它对双缓冲技术提供了支持：生产者与消费者各自维护一个缓冲区，双方通过执行 Exchanger.exchange(V)来交换各自持有的缓冲区。当消费者在“消费”一个已填充完毕的缓冲区时，生产者可以对待填充的缓冲区进行填充（生产产品），从而实现了产品的“消费”与生成的并发。Exchanger 类便于我们能够对产品的粒度进行优化。

Java 线程中断机制相当于 Java 线程与线程间协作的一套协议框架：发起线程通过 Thread.interrupt()调用给目标线程发送中断，这相当于将目标线程的线程中断标记置为 true；目标线程则通过 Thread.currentThread().isInterrupted()/Thread.interrupted()来获取或者获取并重置线程中断标记；发起线程给目标线程发送中断所导致的结果取决于目标线程对中断的响应方式。给目标线程发送中断还能够产生唤醒目标线程的效果。目标线程可以通过对 InterruptedException 进行处理的方式或者直接通过判断线程中断标记并执行相应的处理逻辑的方式来响应中断。对 InterruptedException 进行处理的正确方式包括：不捕获 InterruptedException、捕获 InterruptedException 后重新将该异常抛出，以及捕获 InterruptedException 并在捕获该异常后中断当前线程。

需要主动停止线程的典型场景包括：服务或者系统关闭、错误处理以及用户取消任务。通用的线程优雅停止办法：发起线程更新目标线程的线程停止标记并给其发送中断，目标线程仅在当前无待处理任务且不会产生新的待处理任务情况下才能使 run 方法返回。Web 应用自身启动的工作者线程需要由应用自身在 Web 应用停止时主动停止。



图 5-6 本章知识结构图

第6章

保障线程安全的设计技术

第3章更多的是从Java平台本身提供的机制的角度来介绍如何保障线程安全。本章将从面向对象设计的角度出发介绍几种保障线程安全的常用技术。这些技术的使用通常可以使得我们在不必借助锁的情况下保障线程安全，从而既避免锁可能导致的问题以及开销，又有利于提高系统的并发性并简化代码。另外，本章还介绍了常用的线程安全的集合对象。

*6.1 Java 运行时存储空间

了解Java运行时存储空间的有关知识有助于我们更好地理解多线程编程。Java运行时（Java Runtime）空间可以分为堆（Heap）空间、栈（Stack）空间和非堆（Non-Heap）空间。其中，堆空间和非堆空间是可以被多个线程共享的，而栈空间则是线程的私有空间，每个线程都有其栈空间，并且一个线程无法访问其他线程的栈空间。

堆空间（Heap space）用于存储对象，即创建一个实例的时候该实例所需的存储空间是在堆空间中进行分配的，堆空间本身是在Java虚拟机启动的时候分配的一段可以动态扩容的内存空间。因此，类的实例变量是存储在堆空间中的。由于堆空间是线程之间的共享空间，因此实例变量以及引用型实例变量所引用的对象是可以被多个线程共享的。不管引用对象的变量的作用域如何（局部变量、实例变量和静态变量），对象本身总是存储在堆空间中的。堆空间也是垃圾回收器（Garbage Collector）工作的场所，即堆空间中没有可达引用的对象（不再被使用的对象）所占用的存储空间会被垃圾回收器回收。堆空间通常可以进一步划分为年轻代（Young Generation）和年老代（Old/Tenured Generation）。对象所需的存储空间是在年轻代中进行分配的。垃圾回收器对年轻代中的对象进行的垃圾回收被称为次要回收（Minor Collection）。次要回收中“幸存”下来（即没有被回收掉）的对象最终可能被移入（改变对象所在的存储空间）年老代。垃圾回收器对年老代中的对象进行的垃圾回收被称为主要回收（Major Collection）。

栈空间 (Stack Space) 是为线程的执行而准备的一段固定大小的内存空间, 每个线程都有其栈空间¹。栈空间是在线程创建的时候分配的。线程执行 (调用) 一个方法前, Java 虚拟机会在该线程的栈空间中为这个方法调用创建一个栈帧 (Frame)。栈帧用于存储相应方法的局部变量、返回值等私有数据。可见, 局部变量的变量值存储在栈空间中。基础类型 (Primitive Type) 变量和引用类型 (Reference Type) 变量的变量值都是直接存储在栈帧中的²。引用型变量的值相当于被引用对象的内存地址, 而引用型变量所引用的对象仍然在堆空间中。也就是说, 对于引用型局部变量, 栈帧中存储的是相应对象的内存地址而不是对象本身! 由于一个线程无法访问另外一个线程的栈空间, 因此, 线程对局部变量以及对只能通过当前线程的局部变量才能访问到的对象进行的操作具有固有 (Inherent) 的线程安全性。

非堆空间 (Non-Heap Space) 用于存储常量以及类的元数据 (Meta-data) 等, 它也是在 Java 虚拟机启动的时候分配的一段可以动态扩容的内存空间。类的元数据包括类的静态变量、类有哪些方法以及这些方法的元数据 (包括名称、参数和返回值等)。非堆空间也是多个线程之间共享的存储空间。类的静态变量在非堆空间中的存储方式与局部变量在栈空间的存储方式相似, 即这些空间中仅存储变量的值本身, 而引用型变量所引用的对象仍然存储在堆空间中。

例如, Java 虚拟机运行如清单 6-1 所示的程序所涉及的运行时空间如图 6-1 所示 (图中箭头表示引用型变量对相应对象的引用关系)。

清单 6-1 Java 运行时空间示例代码

```
public class JavaMemory {
    public static void main(String[] args) {
        String msg = args.length > 0 ? args[0] : null;
        ObjectX objX = new ObjectX();
        objX.greet(msg);
    }
}

class ObjectX implements Serializable {
    private static final long serialVersionUID = 8554375271108416940L;
    private static AtomicInteger ID_Generator = new AtomicInteger(0);
    private Date timeCreated = new Date();
    private int id;

    public ObjectX() {
```

1 指这种空间一经分配, 其大小就不可再变大或者变小。

2 基础类型包括 boolean、byte、char、short、int、float、long 和 double。


```

        this.id = ID_Generator.getAndIncrement();
    }

    public void greet(String message) {
        String msg = toString() + ":" + message;
        Debug.info(msg);
    }

    @Override
    public String toString() {
        return "[" + timeCreated + "] ObjectX [" + id + "]";
    }
}

```

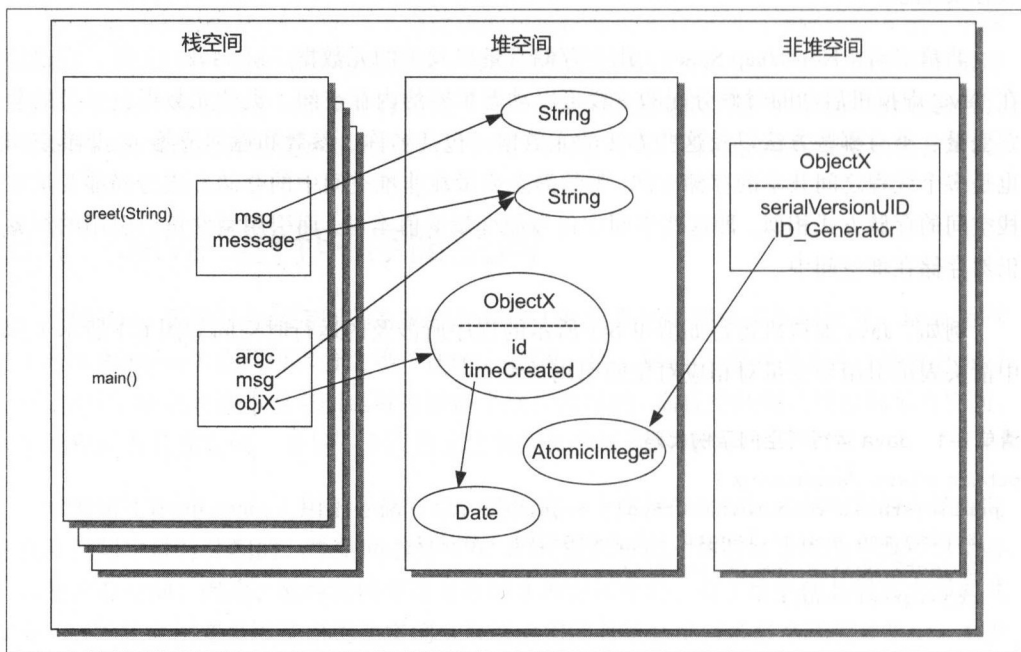


图 6-1 Java 运行时存储空间示意图

提示

堆空间、非堆空间是线程间可共享的空间，这表现为实例变量和静态变量是线程间可共享的；栈空间是线程的私有空间，这表现为局部变量是无法被多个线程共享的。线程对局部变量以及对只能通过当前线程的局部变量才能访问到的对象进行的操作具有固有（Inherent）的线程安全性。

6.2 大公无私：无状态对象

对象（Object）就是操作和数据的封装。对象所包含的数据就被称为该对象的状态（State），它包括存储在实例变量或者静态变量之中的数据。一个对象的状态也可能包含该对象引用的其他对象的实例变量或者静态变量中的数据。相应地，实例变量、静态变量也被称为状态变量（State Variable）。如果一个类的同一个实例被多个线程共享并不会使这些线程存在共享状态（Shared State），那么这个类及其任意一个实例就被称为无状态对象（Stateless Object）。反之，如果一个类的同一个实例被多个线程共享，会使这些线程存在共享状态，那么这个类及其任意一个实例就被称为有状态对象（Stateful Object）。无状态对象不含任何实例变量，且不包含任何静态变量或者其包含的静态变量都是只读的（常量）。有状态对象又可以分为状态可变对象和状态不可变对象。所谓状态可变就是，对象在其生命周期中，其状态变量的值可以发生变化。

我们知道线程安全问题产生的前提是多个线程之间存在共享数据。因此，实现线程安全的一种自然的方法就是避免在多个线程之间共享数据。使用无状态对象就是这样一种自然的办法：一个线程执行无状态对象的任意一个方法来完成某个计算的时候，该计算的瞬时状态（中间结果）仅体现在局部变量和（或）只有当前执行线程能够访问的对象的狀態上。因此，一个线程执行无状态对象的任何方法都不会对访问该无状态对象的其他线程产生任何干扰作用。所以，无状态对象具有固有的线程安全性，它可以被多个线程共享，而这些线程在执行该对象的任何方法时都无须使用同步机制。

下面看一个无状态对象使用实例。假设我们要对第3章的第1个实战案例（负载均衡模块）中的服务器节点（Endpoint类，参见清单3-11）进行排序的话，那么我们可以创建一个Comparator实例来表示相应的排序规则，如清单6-2所示。

清单6-2 无状态对象实例

```
public class DefaultEndpointComparator implements Comparator<Endpoint> {
    @Override
    public int compare(Endpoint server1, Endpoint server2) {
        int result = 0;
        boolean isOnline1 = server1.isOnline();
        boolean isOnline2 = server2.isOnline();
        // 优先按照服务器是否在线排序
        if (isOnline1 == isOnline2) {
            // 被比较的两台服务器都在线（或不在线）的情况下进一步比较服务器权重
            result = compareWeight(server1.weight, server2.weight);
        } else {
            // 在线的服务器排序靠前
            if (isOnline1) {
```

```

        result = -1;
    }
}
return result;
}

private int compareWeight(int weight1, int weight2) {
    // ...
}
}

```

`DefaultEndpointComparator` 的实例就是一个无状态对象：`DefaultEndpointComparator`。`compare` 方法执行时所产生的瞬时状态仅体现为局部变量以及只有执行线程才能访问的对象（`Endpoint` 实例）。在此基础上我们可以实现排序，如清单 6-3 所示。一个 `DefaultEndpointComparator` 实例可以被 `EndpointView.retrieveServerList()` 的多个执行线程共享（通过静态变量 `DEFAULT_COMPARATOR`），而这些线程无须使用锁等同步机制。

清单 6-3 对服务器节点进行排序

```

public class EndpointView {
    static final Comparator<Endpoint> DEFAULT_COMPARATOR;
    static {
        DEFAULT_COMPARATOR = new DefaultEndpointComparator();
    }

    // 省略其他代码

    public Endpoint[] retrieveServerList(Comparator<Endpoint> comparator) {
        Endpoint[] serverList = doRetrieveServerList();
        Arrays.sort(serverList, comparator);
        return serverList;
    }

    public Endpoint[] retrieveServerList() {
        return retrieveServerList(DEFAULT_COMPARATOR);
    }

    private Endpoint[] doRetrieveServerList() {
        // ...
    }

    public static void main(String[] args) {
        EndpointView endpointView = new EndpointView();
        Endpoint[] serverList = endpointView.retrieveServerList();
        Debug.info(Arrays.toString(serverList));
    }
}

```

无状态对象具有线程安全性，这有两层含义：首先，无状态对象的客户端代码在调用该对象的任何方法时都无须加锁。例如 `EndpointView.retrieveServerList()` 在访问 `DefaultEndpointComparator` 实例的时候无须加锁。其次，无状态对象自身的方法实现也无须使用锁。例如，`DefaultEndpointComparator.compare` 方法中没有使用任何锁。

正如清单 6-2 的代码所展示的那样，无状态对象（以及该类的任何一个上层类）是不包含任何实例变量或者任何可更新的静态变量的³。但是，有时候我们可能很难找到一个像 `DefaultEndpointComparator` 实例那样“纯粹”的无状态对象——一个类即使不包含任何实例变量或者静态变量，执行这个类方法的多个线程仍然可能存在共享状态，如清单 6-4 所示。

清单 6-4 多个线程访问本身不包含状态的对象也可能存在共享状态示例

```
public class BrokenStatelessObject {
    public String doSomething(String s) {
        UnsafeSingleton us = UnsafeSingleton.INSTANCE;
        int i = us.doSomething(s);
        UnsafeStatefullObject sfo = new UnsafeStatefullObject();
        String str = sfo.doSomething(s, i);
        return str;
    }

    public String doSomething1(String s) {
        UnsafeSingleton us = UnsafeSingleton.INSTANCE;
        UnsafeStatefullObject sfo = new UnsafeStatefullObject();
        String str;
        synchronized (this) {
            str = sfo.doSomething(s, us.doSomething(s));
        }
        return str;
    }
}

class UnsafeStatefullObject {
    static Map<String, String> cache = new HashMap<String, String>();

    public String doSomething(String s, int len) {
        String result = cache.get(s);
        if (null == result) {
            result = md5sum(result, len);
            cache.put(s, result);
        }
    }
}
```

3 上层类（Superclass）指处于类的继承结构中上层的类，包括一个类的父类以及父类的父类，等等。

```

        return result;
    }

    public String md5sum(String s, int len) {
        // 生成 md5 摘要
        // 省略其他代码
        return s;
    }
}

enum UnsafeSingleton {
    INSTANCE;

    public int state1;

    public int doSomething(String s) {
        // 省略其他代码

        // 访问 state1
        return 0;
    }
}

```

尽管 `BrokenStatelessObject` 类自身不包含任何实例变量或者静态变量，但是 `BrokenStatelessObjec.doSomething` 方法的多个执行线程仍然可能存在共享状态。`BrokenStatelessObjec.doSomething` 方法中使用的 `UnsafeSingleton` 是一个非线程安全单例类（该类仅有一个实例 `UnsafeSingleton.INSTANCE`）。因此，`BrokenStatelessObjec.doSomething` 方法的多个执行线程其实是在共享同一个 `UnsafeSingleton` 实例，而 `UnsafeSingleton` 类的实例变量 `state1` 就成为这些线程的共享状态。尽管 `BrokenStatelessObjec.doSomething` 方法的多个执行线程各自都访问各自的 `UnsafeStatefullObject` 实例，但是 `UnsafeStatefullObject` 的静态变量 `cache` 会成为这些线程的共享状态。因此，即使一个类不包含任何实例变量或者静态变量，执行该类方法的多个线程也仍然可能存在共享状态。此时，这个类在调用其他类的方法时仍然可能需要使用锁。例如，`BrokenStatelessObjec.doSomething` 方法可能需要改写为：

```

public String doSomething(String s) {
    UnsafeSingleton us = UnsafeSingleton.INSTANCE;
    UnsafeStatefullObject sfo = new UnsafeStatefullObject();
    String str;
    synchronized(this){
        str = sfo.doSomething(s,us.doSomething(s));
    }
    return str;
}

```

注意

无状态对象不包含任何实例变量或者可更新静态变量（包括来自相应类的上层类的实例变量或者静态变量）。但是，一个类不包含任何实例变量或者静态变量却不一定无状态对象。特殊情况下，不包含任何实例变量或者静态变量的类，其方法实现时仍然需要借助锁来保障线程安全。

从面向对象编程的角度来看，无状态对象由于不包含任何状态，因此同一个类的多个无状态对象之间是没有差别的。既然如此，我们又为何要使用对象（无状态对象）而不是使用一个仅包括静态方法的类呢？这个问题还是得从面向对象编程中的抽象（Abstraction）与实现（Implementation）这两个层次来回答。例如，`Arrays.sort(T[] a, Comparator<? super T> c)` 允许我们指定一个 `Comparator` 接口实例（`c`）用于指定数组元素的排序规则。这里的 `Comparator` 接口就是对排序规则的抽象，而 `sort` 方法的调用方所传递的具体 `Comparator` 实例则代表实现——一个具体的排序规则。显然，我们在调用这个 `sort` 方法的时候必须传递一个 `Comparator` 接口实现类的一个实例（对象），而无法传递一个类（尽管类本身在 Java 平台中也是一种对象）。

无状态对象可以被多个线程共享，而其客户端代码及其自身的方法实现又无须使用锁，从而避免了锁可能产生的问题（例如死锁）以及开销。因此，无状态对象有利于提高并发性。然而，有时候设计出一个纯粹的无状态对象可能有些难度。另外，即便是纯粹的无状态对象，随着代码的维护，它也可能逐渐演变成其内部实现需要借助锁等线程同步机制的“非纯粹”的无状态对象：无状态对象的一些方法可能在代码维护过程中需要访问一些非线程安全对象，而这些对象的访问可能导致这些方法的执行线程存在共享状态。

实践：正确编写 Servlet 类

无状态对象的一个典型应用就是 Java EE 中的 `Servlet`。`Servlet` 是一个实现 `javax.servlet.Servlet` 接口的托管（`Managed`）类，而不是一个普通的类。所谓托管类，是指 `Servlet` 类实例的创建、初始化以及销毁的整个对象生命周期完全是由 Java Web 服务器（例如 `Tomcat`）控制的，而服务器为每一个 `Servlet` 类最多只生成一个实例，该唯一实例会被用于处理服务器接收到的多个请求。即一个 `Servlet` 类的一个（唯一的）实例会被多个线程共享，并且服务器调用 `Servlet.service` 方法时并没有加锁，因此使 `Servlet` 实例成为无状态对象有利于提高服务器的并发性。这也是 `Servlet` 类一般不包含实例变量或者静态变量的原因：一旦 `Servlet` 类包含实例变量或者静态变量，我们就需要考虑是否使用锁以保障其线程安全。例如，清单 6-5 展示了一个错误（非线程安全）的 `Servlet` 类。该 `Servlet` 类为了避免重复创建 `SimpleDateFormat` 实例的开销而将 `SimpleDateFormat` 实例作为 `Servlet` 类的一个实例变量 `sdf`，然而该 `Servlet` 对 `sdf` 的访问又没有加锁，从而导致 `sdf.parse(String)`

调用解析出来的日期可能是一个客户端根本没有提交过的错误日期！

清单 6-5 非线程安全的 Servlet 示例

```
/**
 * 该类是一个错误的 Servlet 类（非线程安全）
 *
 * @author Viscent Huang
 */
public class UnsafeServlet extends HttpServlet {
    private static final long serialVersionUID = -2772996404655982182L;
    private final SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {

        String strExpiryDate = req.getParameter("expiryDate");
        try {
            sdf.parse(strExpiryDate);
        } catch (ParseException e) {
            e.printStackTrace();
        }
        // 省略其他代码
    }
}
```

6.3 以“不变”应万变：不可变对象

不可变对象（Immutable Object）是指一经创建其状态就保持不变的對象。不可变对象也具有固有的线程安全性，因此不可变对象也可以像无状态对象那样被多个线程共享，而这些线程访问这些共享对象的时候无须加锁。当不可变对象所建模的现实实体的状态发生变化时，系统通过创建新的不可变对象实例来进行反映。

一个严格意义上的不可变对象要同时满足以下所有条件。

- 类本身使用 final 修饰：这是为了防止通过创建子类来改变其定义的行为。
- 所有字段都是用 final 修饰的：使用 final 修饰不仅仅是从语义上说明被修饰字段的值不可改变；更重要的是这个语义在多线程环境下保证了被修饰字段的初始化安全，即 final 修饰的字段在对其他线程可见时，它必定是初始化完成的。
- 对象在此初始化过程中没有逸出（Escape）：防止其他类（如该类的内部匿名类）

在对象初始化过程中修改其状态。

- 任何字段，若其引用了其他状态可变的对象（如集合、数组等），则这些字段必须是 `private` 修饰的，并且这些字段值不能对外暴露。若有相关方法要返回这些字段值，则应该进行防御性复制（Defensive Copy）。

第3章第1个实战案例（负载均衡器）中使用的 `Candidate` 类（参见清单 3-13）就是一个不可变对象，如下代码片段所示：

```
public final class Candidate implements Iterable<Endpoint> {
    // 下游部件节点列表
    private final Set<Endpoint> endpoints;
    // 下游部件节点的总权重
    public final int totalWeight;

    public Candidate(Set<Endpoint> endpoints) {
        int sum = 0;
        for (Endpoint endpoint : endpoints) {
            sum += endpoint.weight;
        }
        this.totalWeight = sum;
        this.endpoints = endpoints;
    }

    @Override
    public final Iterator<Endpoint> iterator() {
        return ReadOnlyIterator.with(endpoints.iterator());
    }
    // 省略其他代码
}
```

`Candidate` 实例的状态包括下游部件的服务器节点列表（`endpoints`）以及这些节点的总权重（`totalWeight`）。如果下游部件的服务器节点需要变更，例如要增加一个服务器节点或者有个节点的权重需要调整，那么，我们需要同时更新服务器节点列表以及相应的总权重。这里所谓的“同时”意味着这个更新操作必须是一个原子操作，否则其他线程可能看到总权重与服务器节点列表中各个节点的权重总和不一致的情形。如果 `Candidate` 类的状态是可变的，那么为了保障这个操作的原子性，我们往往需要借助锁。而在这个案例中，`Candidate` 是个不可变对象，因此这个更新操作通过创建一个新的 `Candidate` 实例并以该实例为参数调用 `AbstractLoadBalancer.updateCandidate` 方法（参见清单 3-12）即可实现。`AbstractLoadBalancer` 类内部会维护一个 `volatile` 实例变量 `candidate` 来引用 `Candidate` 实例，如下代码片段所示：

```
public abstract class AbstractLoadBalancer implements LoadBalancer {
```



```

private final static Logger LOGGER = Logger.getAnonymousLogger();
// 使用 volatile 变量替代锁（有条件替代）
protected volatile Candidate candidate;
protected final Random random;
// 心跳线程
private Thread heartbeatThread;

@Override
public void updateCandidate(final Candidate candidate) {
    if (null == candidate || 0 == candidate.getEndpointCount()) {
        throw new IllegalArgumentException("Invalid candidate " + candidate);
    }
    // 更新 volatile 变量 candidate
    this.candidate = candidate;
}

// 其他代码参见清单 3-12
}

```

这里，`candidate` 实例变量是配置管理线程（负责执行 `updateCandidate` 方法）和业务线程所共享的对象。`volatile` 关键字保障了对实例变量 `candidate` 的写操作的原子性，从而保障整个更新操作（更新下游部件的节点以及总权重）的原子性。另外，`volatile` 关键字还保障了这种更新的结果对于业务线程的可见性。

从上述例子中可以看出，不可变对象可以使我们在无须借助锁的情况下实现线程安全，从而避免了锁可能产生的问题以及开销。

有时创建严格意义上的不可变对象比较难，此时不妨考虑使用等效或者近似的不可变对象，这也同样有利于发挥不可变对象的优势。所谓“等效或者近似”，就是尽可能地满足不可变对象所需的条件。例如，上述案例中涉及的 `Endpoint` 类（参见清单 3-11）就是一个等效不可变对象。

不可变对象的使用能够对垃圾回收效率产生影响，其影响既有消极的也有积极的。由于基于不可变对象的设计中系统状态的变更是通过创建新的不可变对象实例来实现的，因此，当系统的状态频繁变更或者不可变对象所占用的内存空间比较大时，不可变对象的不断创建会增加垃圾回收的负担。但是，不可变对象的使用也可能有利于降低垃圾回收的开销。这是因为创建不可变对象往往导致堆空间年轻代（Young Generation）中的对象（新创建的不可变实例）引用年老代（Old Generation）中的对象。而这种对象引用方式，相比于使用状态可变的对象所导致的年老代对象引用年轻代对象的引用方式，更加有利于减少垃圾回收的开销：修改一个状态可变对象的实例变量值的时候，如果这个对象已经位于年老代中，那么在垃圾回收器进行下一轮次要回收（Minor Collection）的时候，年老代中

包含这个对象的卡片 (Card, 年老代中存储对象的存储单位, 一个 Card 的大小为 512 字节) 中的所有对象都必须被扫描一遍, 以确定年老代中是否有对象对待回收的对象持有引用。因此, 年老代对象持有对年轻代对象的引用会导致次要回收的开销增加。

我们也可以采取某些技术来减少不可变对象 (尤其是比较大的不可变对象) 所占用的内存空间。比如, 创建不可变对象的时候尽可能让新的不可变对象与老的不可变对象共享部分内存空间, 从而减少内存空间占用。在如清单 6-6 所示的例子中, `BigImmutableObject` 的其中一个构造器允许我们指定一个现有的 `BigImmutableObject` 实例 (老的不可变对象) 作为创建新实例的“模板”, 该构造器会调用 `BigImmutableObject.createRegistry` 方法。`BigImmutableObject.createRegistry` 方法会对指定的 `BigImmutableObject` 实例的 `registry` 实例变量进行浅复制 (Swallow Copy) 得到一个新的 `HashMap`, 再对这个新的 `HashMap` 中需要更新的条目进行更新。更新后的 `HashMap` 实例会被作为新 `BigImmutableObject` 实例的 `registry` 实例变量的初始值 (也是最终值)。由于 `BigImmutableObject.createRegistry` 方法所创建的 `HashMap` 实例是老的 `BigImmutableObject` 实例的 `registry` 变量的一个浅复制对象, 因此这两个 `HashMap` 实例会共用大部分存储空间 (主要是 `HashMap` 实例所引用的 `BigObject` 所占用的存储空间)。

清单 6-6 减少不可变对象所占用的空间

```
public final class BigImmutableObject implements
    Iterable<Map.Entry<String, BigObject>> {
    private final HashMap<String, BigObject> registry;

    public BigImmutableObject(HashMap<String, BigObject> registry) {
        this.registry = registry;
    }

    public BigImmutableObject(BigImmutableObject prototype, String key,
        BigObject newValue) {
        this(createRegistry(prototype, key, newValue));
    }

    @SuppressWarnings("unchecked")
    private static HashMap<String, BigObject> createRegistry(
        BigImmutableObject prototype, String key,
        BigObject newValue) {
        // 从现有对象中复制 (浅复制) 字段
        HashMap<String, BigObject> newRegistry =
            (HashMap<String, BigObject>) prototype.registry.clone();
        // 仅更新需要更新的部分
        newRegistry.put(key, newValue);
        return newRegistry;
    }
}
```

```

    }

    @Override
    public Iterator<Entry<String, BigObject>> iterator() {
        // 对 entrySet 进行防御性复制
        final Set<Entry<String, BigObject>> readOnlyEntries = Collections
            .unmodifiableSet(registry.entrySet());

        // 返回一个只读的 Iterator 实例
        return ReadOnlyIterator.with(
            readOnlyEntries.iterator());
    }

    public BigObject getObject(String key) {
        return registry.get(key);
    }

    public BigImmutableObject update(String key,
        BigObject newValue) {
        return new BigImmutableObject(this, key, newValue);
    }
}

class BigObject {
    // 省略其他代码
}

```

基于上述原因，当被建模对象的状态变更比较频繁时，不可变对象也不见得就不能使用。此时，我们需要综合考虑被建模对象的规模、代码目标运行环境的 Java 虚拟机堆内存容量、系统对吞吐率和响应性的要求这几个因素。若这几个方面因素综合考虑都能满足要求，那么使用不可变对象建模也未尝不可。

虽然不可变对象自身的实例变量或者静态变量的值是不可改变的，但是这些变量所引用的对象本身的状态可能是可变的。例如，清单 6-6 所示的例子中 `BigImmutableObject` 的实例变量 `registry` 值是不可变的，但是它所引用的 `HashMap` 对象的状态是可变的（比如可以更新其中一个条目）。此时，这些对象所包含的状态如果需要对外暴露的话，那么我们就需要注意这些对象状态也不能被更改。这通常有两种实现方法。一种是使用迭代器（`Iterator`）模式，即让相应的不可变对象实现 `Iterable` 接口，然后在该接口定义的 `iterator` 方法中返回一个只读的 `Iterator` 实例（它不支持 `remove` 方法）。这样，不可变对象的客户端代码利用 `Iterator` 实例，就可以对相应的不可变对象进行遍历操作，而不必关心也不能更改其内部结构。例如，第 3 章中的 `Candidate` 类（参见清单 3-13）就采用了这种方法来阻止客户端代码更新其实例变量 `endpoints` 所引用的 `Set<Endpoint>` 实例。另外一种方法是

防御性复制 (Defensive Copy)。例如, 清单 6-6 中的 `iterator` 方法除了使用第一种方法创建一个只读的 `Iterator` 实例, 还通过调用 `Collections.unmodifiableSet` 方法来对 `HashMap` 的 `entrySet` 进行防御性复制。

注意

- 当被建模现实实体的状态频繁变化的时候, 不可变对象也不一定就不能使用。
- 不可变对象的使用对垃圾回收效率的影响既有消极的一面, 也有积极的一面。
- 当一个不可变对象需要对外暴露某些状态的时候, 可以使用迭代器 (`Iterator`) 模式和 (或) 防御性复制来阻止客户端代码对其状态进行修改。

不可变对象的典型应用场景

不可变对象特别适用于以下场景。

- **场景一** 被建模对象的状态变化不频繁。正如上述案例所展示的, 这种场景下可以设置一个工作者线程 (例如上述案例中的配置管理线程) 用于在被建模对象状态变化时创建新的不可变对象。而其他线程则仅读取不可变对象的状态。此场景下的一个小技巧是采用 `volatile` 关键字修饰引用不可变对象的变量, 这样既可以避免使用锁 (如 `synchronized`) 又可以保证可见性。第 3 章的第 1 个实战案例 (负载均衡器) 就属于该场景的应用。当然, 上文也提到过被建模对象的状态变化频繁变化的情况下, 也不见得就适合使用不可变对象。
- **场景二** 同时对一组相关的数据进行写操作, 因此需要保证原子性。此场景为了保证操作的原子性, 通常的做法是使用锁。而此时应用不可变对象, 我们既可以保障原子性又可以避免锁的使用, 从而既简化了代码又提高了代码运行效率。第 3 章的第 1 个实战案例 (负载均衡器) 就属于该场景的应用。
- **场景三** 使用不可变对象作为安全可靠的 Map 键 (Key)。设 `someKey` 为任意一个状态可变对象, `someValue` 为任意一个对象, `map` 为 `Map<K,V>` 接口的任意一个实例 (比如 `HashMap` 实例)。在 `map.put(someKey,someValue)` 被调用之后, 如果 `someKey` 的内部状态变化导致 `someKey.hashCode()` 的返回值产生变化, 那么 `map.get(someKey)` 调用将无法返回 `someValue`, 即使在此期间无任何线程执行 `map.remove(someKey)`! 而如果 `someKey` 是一个不可变对象, 那么 `someKey.hashCode()` 返回值恒定, 因此 `map.get(someKey)` 调用总是可以返回 `someValue` (除非中途 `map.remove(someKey)` 被调用过)。因此, 不可变对象非常适宜用作 Map 的键。

6.4 我有我地盘：线程特有对象

如果多个线程需要共享同一个非线程安全对象，那么我们往往需要借助锁来保障线程安全。事实上，我们也可以选择不共享非线程安全对象——对于一个非线程安全对象，每个线程都创建一个该对象的实例，各个线程仅访问各自创建的实例，且一个线程不能访问另外一个线程创建的实例。这种各个线程创建各自的实例，一个实例只能被一个线程访问的对象就被称为线程特有对象（TSO，Thread Specific Object），相对应的线程就被称为该线程特有对象的持有线程。线程特有对象既保障了对非线程安全对象的访问的线程安全，又避免了锁的开销。另外，对于特定类型的线程特有对象，一个线程往往只需要该对象的一个实例，这个实例可以被该线程（同一个线程）所执行的多个方法（包括不同类的方法）共享，因此线程特有对象也有利于减少对象的创建次数。线程特有对象可能是有状态对象，但是由于这个对象并不会被多个线程共享，因此线程特有对象也具有固有的线程安全性。

`ThreadLocal<T>`类相当于线程访问其线程特有对象的代理（Proxy），即各个线程通过这个对象可以创建并访问各自的线程特有对象，其类型参数 `T` 指定了相应线程特有对象的类型。一个线程可以使用不同的 `ThreadLocal` 实例来创建并访问其不同的线程特有对象。多个线程使用同一个 `ThreadLocal<T>`实例所访问到的对象是类型 `T` 的不同实例，即这些线程各自的线程特有对象实例。因此，`ThreadLocal` 类也可以理解为当前线程访问其线程特有对象的代理对象，这种代理与被代理的关系如图 6-2 所示。

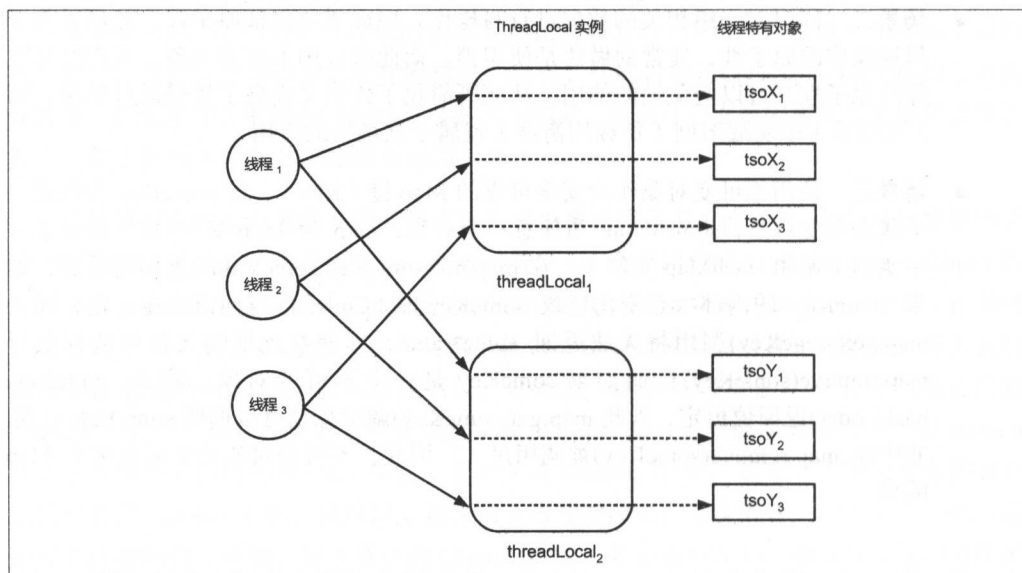


图 6-2 ThreadLocal 与线程特有对象的代理关系示意图

从图 6-2 可以看出, ThreadLocal 实例为每个访问它的线程(即当前线程)都关联了一个该线程的线程特有对象。换句话说,每个 ThreadLocal<T>实例都有一个(且只有一个)当前线程的线程特有对象 T 的实例与之关联,这种关联关系就像一个变量总是有一个(且只有一个)值与之关联一样(尽管变量的值是可以改变的),因此 ThreadLocal 实例也被称为线程局部变量(Thread-local Variable)。ThreadLocal 类的方法如表 6-1 所示。

表 6-1 ThreadLocal 类的常用方法

方 法	功 能
public T get()	获取与该线程局部变量关联的当前线程的线程特有对象
public void set(T value)	重新关联该线程局部变量所对应的当前线程的线程特有对象
protected T initialValue()	该方法的返回值(对象)就是初始状态下该线程局部变量所对应的当前线程的线程特有对象
public void remove()	删除该线程局部变量与相应的当前线程的线程特有对象之间的关联关系

设 tlVar 为任意一个线程局部变量。初始状态下, tlVar 并没有与之关联的线程特有对象。当一个线程初次执行 tlVar.get()的时候, tlVar.get()会调用 tlVar.initialValue()。tlVar.initialValue()的返回值就会成为 tlVar 所关联的当前线程(即 tlVar.get()的执行线程)的线程特有对象。这个线程后续再次执行 tlVar.get()所返回的线程特有对象始终都是同一个对象(即 tlVar.initialValue()的返回值),除非这个线程中途执行了 tlVar.set(T)。由于 ThreadLocal 的 initialValue 方法的返回值为 null,因此要设置线程局部变量关联的初始线程特有对象。我们需要创建 ThreadLocal 的子类(通常是匿名子类)并在子类中覆盖(Override)initialValue 方法,然后在该方法中返回初始线程特有对象。从 Java 8 开始, ThreadLocal 引入了一个名为 withInitial 的静态方法,该方法使得我们能够用一个 Lambda 表达式(返回值)作为相应线程局部变量所关联的初始线程特有对象。例如,清单 6-7 中的线程局部变量 SDF 的初始值可写作 ThreadLocal.withInitial(() -> new SimpleDateFormat("yyyy-MM-dd"))。

使用 ThreadLocal,我们可以将清单 6-5 中的非线性安全 Servlet 改造成线程安全的,如清单 6-7 所示。在这个例子中, ThreadLocal 不仅使我们在无须借助锁的情况下实现了线程安全,还减少了对对象创建的次数——doPost 方法的各个执行线程各自仅创建各自的一个 SimpleDateFormat 实例。相反,如果我们直接在 doPost 方法中创建并使用 SimpleDateFormat 实例的话固然可以确保线程安全,但是那样就意味着每次执行 doPost 方法都会导致新的 SimpleDateFormat 实例被创建。

清单 6-7 使用 ThreadLocal 实现线程安全示例代码

```
public class ServletWithThreadLocal extends HttpServlet {
```

```

final static ThreadLocal<SimpleDateFormat>
SDF = new ThreadLocal<SimpleDateFormat>() {
    @Override
    protected SimpleDateFormat initialValue() {
        return new SimpleDateFormat("yyyy-MM-dd");
    }
};

@Override
protected void doPost(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException {
    final SimpleDateFormat sdf = SDF.get();
    String strExpiryDate = req.getParameter("expiryDate");
    try (PrintWriter pwr = resp.getWriter()) {
        sdf.parse(strExpiryDate);
        // 省略其他代码
        pwr.printf("[%s]expiryDate:%s", Thread.currentThread().getName(),
            strExpiryDate);
    } catch (ParseException e) {
        throw new ServletException(e);
    } // try 结束
}
}

```

线程局部变量通常是会被声明为某个类的静态变量，正如清单 6-7 所示。这是因为：如果把线程局部变量声明为某个类的实例变量，那么每创建该类的一个实例都会导致新的 `ThreadLocal` 实例被创建。这就可能导致当前线程中同一个类型的线程特有对象会被多次创建。而这即便不会导致错误，也会导致重复创建对象带来的浪费。

注意 | `ThreadLocal` 实例通常会被作为某个类的静态字段使用。

由于线程安全的对象内部往往需要使用锁，因此，多个线程共享线程安全的对象可能导致锁的争用。所以，有时候为了避免锁的争用导致的开销（主要是上下文切换），我们也特意将线程安全的对象作为线程特有对象来使用，从而既避免了锁的开销，又减少了对象创建的次数。

下面看一个 `ThreadLocal` 实战案例。某系统会在用户执行某些关键操作前通过短信验证码（一个 6 位数字组成的字符串）来验证操作者的身份，以确定是否是用户本人进行操作的（而不是他人冒充进行操作的）。这个验证码是随机生成的，为了尽量保障这个验证码的随机性，我们使用强随机数生成器 `java.security.SecureRandom`（它是 `Random` 的一个子类）。尽管 `SecureRandom` 是线程安全的，并因此可以被多个线程共享，但是为了避免多个线程共享 `SecureRandom` 实例可能导致的对 `SecureRandom` 内部所使用锁的争用，我们

决定不在多个线程间共享同一 `SecureRandom` 实例。另外，考虑到每次生成验证码的时候都创建一个 `SecureRandom` 也是不现实的（开销太大），因此我们决定将 `SecureRandom` 实例作为一个线程特有对象来使用。该案例中用于生成验证码的随机数生成器如清单 6-8 所示。

清单 6-8 使用 `ThreadLocal` 避免锁的争用

```
public enum ThreadSpecificSecureRandom {
    INSTANCE;

    final static ThreadLocal<SecureRandom> SECURE_RANDOM
        = new ThreadLocal<SecureRandom>() {
        @Override
        protected SecureRandom initialValue() {
            SecureRandom srnd;
            try {
                srnd = SecureRandom.getInstance("SHA1PRNG");
            } catch (NoSuchAlgorithmException e) {
                srnd = new SecureRandom();
                e.printStackTrace();
            }

            // 通过以下调用来初始化种子
            srnd.nextBytes(new byte[20]);
            return srnd;
        }
    };

    // 生成随机数
    public int nextInt(int upperBound) {
        SecureRandom secureRnd = SECURE_RANDOM.get();
        return secureRnd.nextInt(upperBound);
    }

    public void setSeed(long seed) {
        SecureRandom secureRnd = SECURE_RANDOM.get();
        secureRnd.setSeed(seed);
    }
}
```

`ThreadSpecificSecureRandom` 通过线程局部变量来引用 `SecureRandom` 实例，这使得执行 `nextInt` 方法以生成验证码的多个线程各自使用各自的 `SecureRandom` 实例，从而避免了

锁的争用⁴。

JDK 1.7 中引入的标准库类 `java.util.concurrent.ThreadLocalRandom` 的初衷与该案例所要实现的目标相似。`ThreadLocalRandom` 也是 `Random` 的一个子类，它相当于 `ThreadLocal<Random>`。不过，`ThreadLocalRandom` 所产生的随机数并非强随机数。

6.4.1 线程特有对象可能导致的问题及其规避

使用线程特有对象可能会导致如下几个问题。

- 退化与数据错乱。由于线程和任务之间可以是一对多的关系，即一个线程可以先后执行多个任务，因此线程特有对象就相当于一个线程所执行的多个任务之间的共享对象。如果线程特有对象是个有状态对象且其状态会随着相应线程所执行的任务而改变，那么这个线程所执行的下一个任务可能“看到”来自前一个任务的数据，而这个数据可能与该任务并不匹配，从而导致数据错乱。因此，在一个线程可以执行多个任务的情况下（比如在生产者-消费者模式中）使用线程特有对象，我们需要确保每个任务的处理逻辑被执行前相应的线程特有对象的状态不受前一个被执行的任务影响。这通常可以通过在任务处理逻辑被执行前为线程局部变量重新关联一个线程特有对象（通过调用 `ThreadLocal.set(T)` 实现）或者重置线程特有对象的状态来实现。例如，清单 6-9 中的 `XAbstractTask` 子类的多个实例可以由一个线程负责执行（比如使用第 5 章的 `TaskRunner` 来执行，代码参见清单 5-14），因此我们在 `preRun` 方法中将线程特有对象 `HashMap` 的内容清空，以避免前一个任务（`XAbstractTask` 子类实例）执行时更改了线程特有对象的状态对当前执行的任务造成影响。从清单 6-9 中可以看出，在线程可以被重复使用来执行多个任务的情况下使用线程特有对象即使不会造成数据错乱，也可能导致这种线程特有对象实际上“退化”成为任务特有对象——被执行的任务可能更改了线程特有对象的状态，而这些状态一旦对其他任务可见又可能导致数据错乱，因此每个任务实际上需要的是状态会受该任务影响并且独立于其他任务的一个对象。

清单 6-9 避免 `ThreadLocal` 可能导致的数据错乱

```
public abstract class XAbstractTask implements Runnable {
    static ThreadLocal<HashMap<String, String>> configHolder = new
        ThreadLocal<HashMap<String, String>>() {
        @Override
```

4 由于 `SecureRandom` 的内部实现可能涉及多个 `SecureRandom` 实例从同一个熵池（Entropy Pool）中获取随机数生成器所需的种子（Seed），因此系统中创建的 `SecureRandom` 实例越多则熵池中熵（Entropy）不够用的概率就越大，当系统中的熵不够用时，那么获取熵的线程就会被阻塞。因此，在工作者线程数较大的情况下以线程特有对象的方式来使用 `SecureRandom` 需要注意系统中熵的数量的有限性。

```

protected HashMap<String, String> initialValue() {
    return new HashMap<String, String>();
}

};

// 该方法总是会在任务处理逻辑被执行前执行
protected void preRun() {
    // 清空线程特有对象 HashMap 实例, 以保证每个任务执行前 HashMap 的内容是“干净”的
    configHolder.get().clear();
}

protected void postRun() {
    // 什么也不做
}

// 暴露给子类用于实现任务处理逻辑
protected abstract void doRun();

@Override
public final void run() {
    try {
        preRun();
        doRun();
    } finally {
        postRun();
    }
}
}

```

- ThreadLocal 可能导致内存泄漏、伪内存泄漏。在 Web 应用中使用 ThreadLocal 极易导致内存泄漏、伪内存泄漏的问题。下面以 Tomcat 服务器环境为例分析 ThreadLocal 可能导致内存泄漏、伪内存泄漏的原因,并在此基础上给出规避措施。

术语定义

内存泄漏 (Memory Leak) 指由于对象永远无法被垃圾回收导致其占用的 Java 虚拟机内存无法被释放。持续的内存泄漏会导致 Java 虚拟机可用内存逐渐减少,并最终可能导致 Java 虚拟机内存溢出 (Out of Memory),直到 Java 虚拟机宕机。

伪内存泄漏 (Memory Pseudo-leak) 类似于内存泄漏。所不同的是,伪内存泄漏中对象所占用的内存存在其不再被使用后的相当长时间仍然无法被回收,甚至可能永远无法被回收。也就是说,伪内存泄漏中对象占用的内存空间可能会被回收,也可能永远无法被回收 (此时,就变成了内存泄漏)。

我们先简单了解一下 ThreadLocal 的内部实现机制。在 Java 平台中,每个线程 (Thread 实例) 内部会维护一个类似 HashMap 的对象,我们称之为 ThreadLocalMap。每个

ThreadLocalMap 内部会包含若干 Entry（条目，一个键 Key-值 Value 对）。因此，我们可以说每个线程都拥有若干这样的条目，相应的线程就被称为这些条目的属主线程。Entry 的 Key 是一个 ThreadLocal 实例，Value 是一个线程特有对象。因此，Entry 的作用相当于为其属主线程建立起一个 ThreadLocal 实例与一个线程特有对象之间的对应关系。由于 Entry 对 ThreadLocal 实例的引用（通过 Key 引用）是一个弱引用（Weak Reference），因此它不会阻止被引用的 ThreadLocal 实例被垃圾回收。当一个 ThreadLocal 实例没有对其可达的（Reachable）强引用时，这个实例可以被垃圾回收，即其所在的 Entry 的 Key 会被置为 null。此时，相应的 Entry 就成为无效条目（Stale Entry）。另一方面，由于 Entry 对线程特有对象的引用是强引用，因此如果无效条目本身有对它的可达强引用，那么无效条目也会阻止其引用的线程特有对象被垃圾回收。有鉴于此，当 ThreadLocalMap 中有新的 ThreadLocal 到线程特有对象的映射（对应）关系被创建（相当于有新的 Entry 被添加到 ThreadLocalMap）的时候，ThreadLocalMap 会将无效条目清理掉⁵，这打破了无效条目对线程特有对象的强引用，从而使相应的线程特有对象能够被垃圾回收。但是，这个处理也有一个缺点——一个线程访问过线程局部变量之后如果该线程有对其可达的强引用，并且该线程在相当长时间内（甚至一直）处于非运行状态，那么该线程的 ThreadLocalMap 可能就不会有任何变化，因此相应的 ThreadLocalMap 中的无效条目也不会被清理，这就可能导致这些线程的各个 Entry 所引用的线程特有对象都无法被垃圾回收，即导致了伪内存泄漏。

线程对象对 ThreadLocal 和线程特有对象的引用关系如图 6-3 所示（图中虚线表示弱引用，实线表示强引用）。

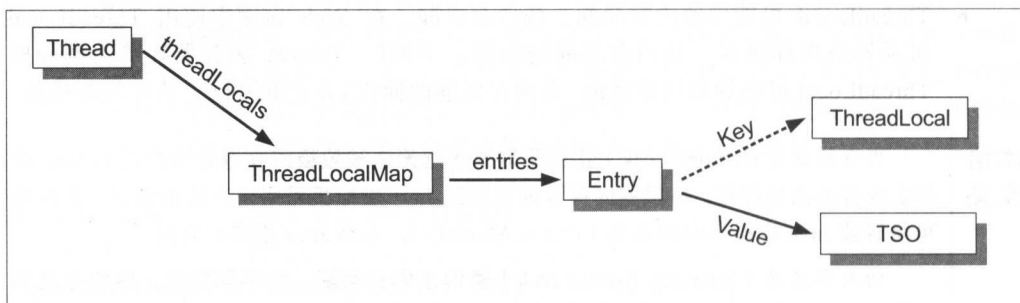


图 6-3 线程与 ThreadLocal、线程特有对象的引用关系

清单 6-10 展示了一个使用 ThreadLocal 并可能导致内存泄漏的 Servlet。

5 ThreadLocalMap 在创建新的 ThreadLocal 到线程特有对象的映射关系时可以复用无效条目，而不一定要创建新的条目来表示这种对应关系。

清单 6-10 ThreadLocal 内存泄漏示例代码

```

/**
 * 该类可能导致内存泄漏!
 * @author Viscent Huang
 */
@WebServlet("/memoryLeak")
public class ThreadLocalMemoryLeak extends HttpServlet {
    private static final long serialVersionUID = 4364376277297114653L;
    final static ThreadLocal<Counter> counterHolder = new ThreadLocal<Counter>() {
        @Override
        protected Counter initialValue() {
            Counter tsoCounter = new Counter();
            return tsoCounter;
        }
    };

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
        doProcess(req, resp);
        try (PrintWriter pwr = resp.getWriter()) {
            pwr.printf("Thread %s,counter:%d",
                Thread.currentThread().getName(),
                counterHolder.get().getAndIncrement());
        }
    }

    void doProcess(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
        counterHolder.get().getAndIncrement();
        // 省略其他代码
    }
}

// 非线性安全
class Counter {
    private int i = 0;
    public int getAndIncrement() {
        return i++;
    }
}

```

在 Tomcat 环境下, Web 应用自身定义的类 (Custom Class) 由类加载器 (Class Loader) WebAppClassLoader 负责加载, 而 Java 标准库类 (例如 HashMap) 由类加载器 StandardClassLoader 负责加载。每个类 (类本身也是一种对象) 都会持有对加载该类的类加载器的强引用, 并且类加载器本身又会持有其加载过的所有类的强引用。另外, 每个对

象（实例）都会持有对其相应类的强引用。由于 Servlet 类 `ThreadLocalMemoryLeak` 及其使用的线程特有对象 `Counter` 类都是由 `WebAppClassLoader` 负责加载的，并且 `counterHolder`（`ThreadLocal<Counter>`）是 `ThreadLocalMemoryLeak` 的一个静态字段，因此我们可以得出图 6-4 所示的引用关系（图中实线表示强引用）。

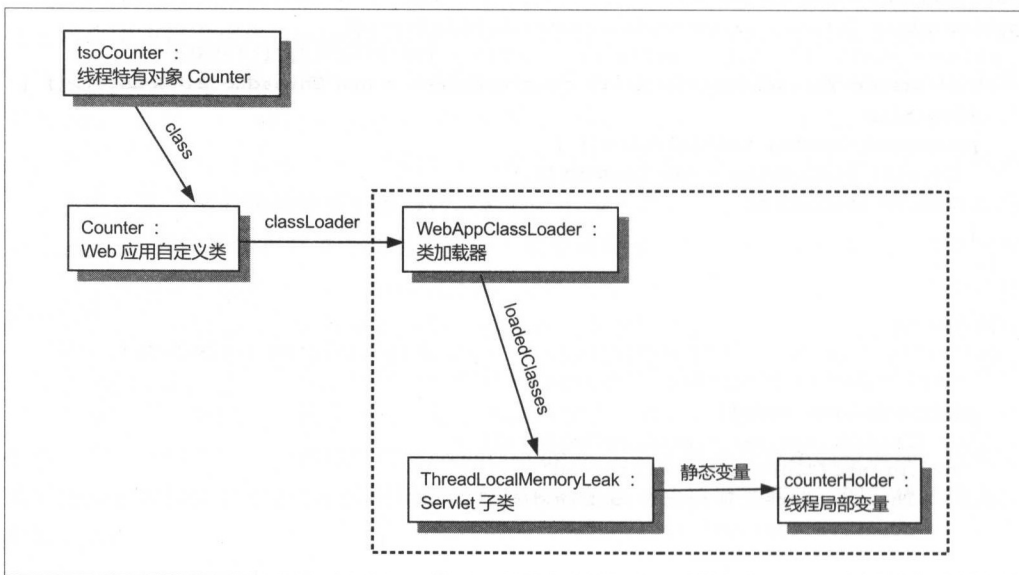


图 6-4 线程特有对象与 `ThreadLocal` 的引用关系

从图 6-4 中可以看出，由 Web 应用自身定义的线程特有对象（`tsoCounter`）持有对线程局部变量（`counterHolder`）的可达引用。并且线程（对象）又持有对其线程特有对象的可达引用（如图 6-3 所示），因此，结合图 6-3、图 6-4 中的引用关系可知，此时线程（对象）不仅持有了对其线程特有对象的可达强引用（见图 6-3），它还通过其线程特有对象持有了对线程局部变量的可达强引用（见图 6-4）。所以，只要系统中还存在对这个线程对象的可达强引用，即线程本身没有被垃圾回收掉，那么这个线程访问过的所有线程局部变量以及相应的线程特有对象都不会被垃圾回收掉！由于 Tomcat 中的一个工作者线程（负责调用 `Servlet.service` 方法进行请求处理，`service` 方法最终会调用 `doXXX` 方法）可以为多个 Web 应用服务，因此当 `ThreadLocalMemoryLeak` 所在的 Web 应用被停止的时候（不是 Web 服务器被停止）执行过 `ThreadLocalMemoryLeak.service` 方法的工作者线程并不会被停止，故而这些线程对象并不会被垃圾回收掉，进而使其所引用的所有线程局部变量及相应的线程特有对象也不会被垃圾回收掉，即导致了内存泄漏。进一步来说，此时的内存泄漏还会导致与当前 Web 应用相应的类加载器 `WebAppClassLoader` 所加载的所有类（以及

这些类的静态变量所引用的所有对象)都无法被垃圾回收,而这最终可能导致 Java 虚拟机的非堆内存 (Non-heap) 空间中的永久代 (Permanent Generation, Java 8 中它被元数据空间 Metaspace 所取代) 内存溢出 (Out of memory), 即 Java 虚拟机会抛出 `java.lang.OutOfMemoryError` (具体消息为 “PermGen space”)。所幸的是, Apache Tomcat 以及 IBM WebSphere Application Server 都提供了一套内存泄漏的检查机制以及一定程度的自动规避机制 (不过, 我们最好不要依赖于这种自动规避机制)⁶。

如果线程局部变量关联的线程特有对象是一个 Java 标准库类 (如清单 6-7 所使用的 `java.text.SimpleDateFormat` 实例), 那么由于 Java 标准库类是由类加载器 `StandardClassLoader` 加载的, `StandardClassLoader` 并不会持有对应用自身定义的类 (`ThreadLocalMemoryLeak`) 的引用, 因此图 6-4 所示的引用关系中虚线框中的引用关系并不存在, 即导致上述内存泄漏的前提不满足。所以, 线程局部变量关联的线程特有对象类型如果是 Java 标准库类, 那么它并不会导致内存泄漏。但是, 由于图 6-3 中的引用关系——线程 (对象) 持有对线程特有对象 (TSO) 的可达强引用, 因此只要相应的线程 (对象) 没有被垃圾回收掉, 那么相应的线程特有对象也不会被垃圾回收掉。可见, 这种情形下, 线程局部变量可能导致伪内存泄漏。

由于 `ThreadLocal` 可能导致内存泄漏、伪内存泄漏的最小前提是线程 (对象) 持有对线程特有对象的可达强引用 (见图 6-3 中的实线所表示的引用关系)。因此, 我们只要打破这种引用, 即通过在当前线程中调用 `ThreadLocal.remove()` 将线程特有对象从其所属的 `Entry` 中剥离 (清理), 便可以使线程特有对象以及线程局部变量都可以被垃圾回收。如果我们仅仅是打破线程特有对象对 `ThreadLocal` 的引用关系 (如图 6-4 所示), 那么只有线程局部变量可以被垃圾回收, 而伪内存泄漏仍然存在, 即线程特有对象可能仍然无法被垃圾回收。

对于同一个 `ThreadLocal` 实例, `ThreadLocal.remove()` 能够奏效的前提是, 其执行线程与 `ThreadLocal.get()/set(T)` 的执行线程必须是同一个线程。由于 `ThreadLocal.get()/remove()/set(T)` 这几个方法都是针对当前线程 (即这些方法的执行线程) 的, 因此即使是针对同一个 `ThreadLocal` 实例, 我们也无法通过在一个线程中调用 `ThreadLocal.remove()` 来将另外一个线程的线程特有对象从其所属的 `Entry` 中剥离。换言之, 我们无法通过在一个线程中执行 `ThreadLocal.remove()` 来规避另外一个线程因使用 `ThreadLocal` 而导致的伪内存泄露⁷!

6 参见: <https://wiki.apache.org/tomcat/MemoryLeakProtection#customThreadLocal> 和 http://www.ibm.com/support/knowledgecenter/en/SS7K4U_8.5.5/com.ibm.websphere.zseries.doc/ae/ctrb_memleakdetection.html。

7 尽管如此, 一个线程通过执行某些操作来规避其他线程因使用 `ThreadLocal` 而导致的内存泄漏的办法还是有可能的。参见本系列图书的“设计模式篇”。

因此，在 Web 应用中，为了规避 ThreadLocal 可能导致的内存泄漏、伪内存泄漏，我们通常需要在 javax.servlet.Filter 接口实现类的 doFilter 方法中调用 ThreadLocal.remove()。这其实是利用 Filter 的一个重要特征（以便 ThreadLocal.remove()调用能够奏效）：Web 服务器对同一个 HTTP 请求进行处理时，Filter.doFilter 方法的执行线程与 Servlet 的执行线程（即 Servlet.service 方法的执行线程）是同一个线程。

如清单 6-11 所示的 Filter 可用于规避 ThreadLocalMemoryLeak 类（见清单 6-10）中定义的线程局部变量 counterHolder 可能导致的内存泄漏、伪内存泄漏。

清单 6-11 使用 Filter 规避 ThreadLocal 内存泄漏示例代码

```
@WebFilter("/memoryLeak")
public class ThreadLocalCleanupFilter implements Filter {
    @Override
    public void doFilter(ServletRequest request, ServletResponse response,
        FilterChain chain)throws IOException, ServletException {
        chain.doFilter(request, response);
        ThreadLocalMemoryLeak.counterHolder.remove();
    }
    // 省略其他代码
}
```

在上述 Filter.doFilter 方法中，我们在 FilterChain.doFilter 调用之后，即请求处理结束之后调用 ThreadLocal.remove()。

在 Web 应用中使用线程特有对象可能导致线程特有对象的“退化”：在上述例子中，为了避免 ThreadLocal 导致的伪内存泄漏（或内存泄漏），我们在每个请求处理结束后都将该请求的处理线程的线程特有对象（Counter 实例）清理掉。因此，不同的请求即使是先后由同一个（任意的）服务器工作者线程来负责处理的，这个（任意的）线程每次执行 ThreadLocalMemoryLeak.doGet 方法（以对请求进行处理）的时候都会创建新的 Counter 实例。这就意味着：首先，不同的服务器工作者线程不会访问相同的 Counter 实例，即 Counter 实例不会被多个服务器工作者线程共享，这说明该例子对 Counter 的使用方式（线程局部变量）与直接将 Counter 实例定义为一个静态变量（final static Counter COUNTER = new Counter();）还是不同的。其次，这些服务器工作者线程所访问的线程特有对象（Counter 实例）实际上已“退化”成“请求特有对象”——每一个请求都对应一个 Counter 实例。

6.4.2 线程特有对象的典型应用场景

典型应用场景如下。

- **场景一** 需要使用非线性安全对象，但又不希望因此而引入锁。如果多个线程需要使用非线性安全的对象，而我们又不希望该对象被多个线程共享（因为共享往往意味着需要引入锁以保证线程安全），此时可以使用线程特有对象，使得各个线程拥有其特有的非线性安全对象实例。清单 6-7 所示的例子就属于该场景的应用。
- **场景二** 使用线程安全对象，但希望避免其使用的锁的开销和相关问题。线程安全的对象虽然可以被多个线程共享，但是由于其可能使用了锁来保证线程安全，而某些情况下我们可能不希望看到锁的开销以及由锁可能引起的相关问题（如死锁）。此时，我们可以将线程安全的对象当作非线性安全的对象来看待。因此，这种场景就转化成场景一。只不过此时使用线程特有对象的主要意图在于避免锁的开销，当然线程安全也是有保障的。如清单 6-8 所示的代码展示了这种使用场景。
- **场景三** 隐式参数传递（Implicit Parameter Passing）。线程特有对象在一个具体的线程中，它是线程全局可见的。一个类的方法中设置的线程特有对象对于该方法调用的任何其他方法（包括其他类的方法）都是可见的。这就可以形成隐式传递参数的效果，即一个类的方法调用另一个类的方法时，前者向后者传递数据可以借助 ThreadLocal 而不必通过方法参数传递。不过，也有的观点认为隐式参数传递使得系统难于理解。隐式参数传递的实现通常是使用一个只包括静态方法的类或者单例类（包装类）来封装对线程特有对象的访问，其他相应访问线程特有对象的代码只需要调用包装类的静态方法或者实例方法即可以访问线程特有对象。
- **场景四** 特定于线程的单例（Singleton）模式。广为使用的单例模式所实现的效果是在一个 Java 虚拟机中的一个类加载器下某个类有且仅有一个实例。如果我们希望对于某个类每个线程有且仅有该类的一个实例，那么就可以使用线程特有对象。例如，在如清单 6-7 所示的代码中，doPost 方法的多个执行线程各自只会创建一个 SimpleDateFormat 实例。

6.5 装饰器模式

装饰器（Decorator）模式可以用来实现线程安全，其基本思想是为非线性安全对象创建一个相应的线程安全的外包装对象（Wrapper），客户端代码不直接访问非线性安全对象而是访问其外包装对象。外包装对象与相应的非线性安全对象具有相同的接口，因此客户端代码使用外包装对象的方式与直接使用相应的非线性安全对象的方式相同，而外包装对象内部通常会借助锁，以线程安全的方式调用相应非线性安全对象的同签名方法来实现其对外暴露的各个方法。

java.util.Collections.synchronizedX（其中，X 可以是 Set、List、Map 等）方法就是使

用装饰器模式将指定的非线性安全的集合对象对外暴露为线程安全的对象（外包装对象）。`Collections.synchronizedX` 方法的参数允许我们指定一个非线性安全的集合对象，该方法的返回值是指定集合对象的外包装对象。这些对象也被称为同步集合（`Synchronized Collection`）。例如，`Collections.synchronizedMap` 方法可以根据指定的非线性安全 `Map` 接口实现类（比如 `HashMap`）返回一个相应的外包装对象（同样也是 `Map` 接口实例）。

使用装饰器模式来实现线程安全的一个好处就是关注点分离（`Separation of Concern`）。在这种设计中，实现同一组功能的对象有两个版本——非线性安全版和线程安全版。例如，对于 `Map` 接口定义的功能，我们有一个非线性安全版的 `HashMap` 和一个线程安全版的 `Collections.synchronizedMap(new HashMap())`。这会带来若干好处：首先，这使得我们可以根据实际需要选择最合适的实现类。比如，如果只有一个线程需要使用 `Map` 接口，那么我们可以选择 `HashMap`，这样可以避免相应的同步集合（`Collections.synchronizedMap` 的返回值）中使用的锁所产生的开销。其次，在非线性安全版的类里我们可以只关注功能本身，而不必关注线程安全问题，即我们能够以单线程的方式去开发非线性安全版的类。这不仅降低了开发难度，还提高了可测试性。而线程安全版的类仅需要关注线程安全问题；至于功能部分，它可以委托给相应的非线性安全版的类，即通过调用相应非线性安全版类的相应方法来实现功能，这同样也能够提高可测试性。

使用装饰器模式来实现线程安全也存在一些缺点，例如 `Collections.synchronizedX` 方法返回的同步集合存在如下弊端。

首先，这些同步集合的 `iterator` 方法返回的 `Iterator` 实例并不是线程安全的。为了保障对同步集合的遍历操作的线程安全性，我们需要对遍历操作进行加锁，如清单 6-12 所示。

清单 6-12 保障对外包装对象的遍历操作的线程安全

```
public class SyncCollectionSafeTraversal {
    final List<String> syncList = Collections.synchronizedList(new
        ArrayList<String>());

    // ...

    public void dump() {
        Iterator<String> iterator = syncList.iterator();
        synchronized (syncList) {
            while (iterator.hasNext()) {
                System.out.println(iterator.next());
            }
        }
    }
}
```

从清单 6-12 中可以看出，对同步集合进行遍历操作的时候，我们需要以被遍历同步集合对象本身作为内部锁。这样做实质上是利用了内部锁的排他性，从而阻止了遍历过程中其他线程改变了同步集合的内部结构。因此，这种遍历是不利于提高并发性的。另外，对遍历操作进行加锁时，我们选用的内部锁必须和相应的同步集合内部用于保障其自身线程安全所使用的锁保持一致。也就是说，这一定程度上要求我们必须知道同步集合对象内部的一些细节，显然这是有悖于面向对象编程中的信息封装（Information Hiding）原则的。

其次，这些同步集合在其实现线程安全的时候通常是使用一个粗粒度的锁，即使用一个锁来保护其内部所有的共享状态。因此，使用这些同步集合虽然可以确保线程安全，但是也可能导致锁的高争用，从而导致较大的上下文切换的开销。

6.6 并发集合

JDK 1.5（以及其后版本）的 `java.util.concurrent` 包中引入了一些线程安全的集合对象，它们被称为并发集合。这些对象通常可以作为同步集合的替代品，它们与常用的非线程安全集合对象之间的对应关系如表 6-2 所示。

表 6-2 常用非线程安全集合对象对应的线程安全对象

非线程安全对象	并发集合类	共同接口	遍历实现方式
ArrayList	CopyOnWriteArrayList	List	快照
HashSet	CopyOnWriteArraySet	Set	快照
LinkedList	ConcurrentLinkedQueue	Queue	准实时
HashMap	ConcurrentHashMap	Map	准实时
TreeMap	ConcurrentSkipListMap	SortedMap	准实时
TreeSet	ConcurrentSkipListSet	SortedSet	准实时

并发集合对象自身就支持对其进行线程安全的遍历操作。应用代码对并发集合对象进行遍历的时候无须加锁就可以实现遍历操作的线程安全⁸。并且，对并发集合的遍历操作和对其进行更新的更新操作是可以由不同的线程并发执行的，从而有利于充分提高系统的并发性。

并发集合实现线程安全的遍历通常有两种方式。一种是对遍历对象的快照进行遍历。快照（Snapshot）是在 `Iterator` 实例被创建的那一刻待遍历对象内部结构的一个只读

8 实际上对于某些并发集合对象（比如 `ConcurrentHashMap`）而言，应用代码也无法在遍历的时候对其进行加锁，以阻止其他线程更改被遍历的对象的内部结构。

副本（对象），它反映了待遍历集合的某一时刻（即 `Iterator` 实例被创建的那一刻）的状态（不包括集合元素的状态）。由于对同一个并发集合进行遍历操作的每个线程会得到各自的一份快照，因此快照相当于这些线程的线程特有对象。所以，这种方式下进行遍历操作的线程无须加锁就可以实现线程安全。另外，由于快照是只读的，因此这种遍历方式所返回的 `Iterator` 实例是不支持 `remove` 方法的⁹。这种方式的优点是遍历操作和更新操作之间互不影响（因为各自操作的是不同的对象），缺点是当被遍历的集合比较大时，创建快照的直接或者间接开销会比较大。`CopyOnWriteArrayList` 和 `CopyOnWriteArraySet` 就使用这种遍历方式。另一种是对待遍历对象进行准实时的遍历。所谓准实时是指遍历操作不是针对待遍历对象的副本进行的，但又不借助锁来保障线程安全，从而使得遍历操作可以与更新操作并发进行。并且，遍历过程中其他线程对被遍历对象的内部结构的更新（比如删除了一个元素）可能会（也可能不会）被反映出来。这种遍历方式所返回的 `Iterator` 实例可以支持 `remove` 方法。`ConcurrentLinkedQueue` 和 `ConcurrentHashMap` 等并发集合就采用这种遍历方式。由于 `Iterator` 是被设计用来一次只被一个线程使用的，因此如果有多个线程需要进行遍历操作，那么这些线程之间是不适宜共享同一个 `Iterator` 实例的！

注意

如果有多个线程需要对同一并发集合进行遍历操作，那么这些线程不适合共享同一个 `Iterator` 实例。

另外，并发集合内部在保障其线程安全的时候通常不借助锁，而是使用 CAS 操作（参见本书第 3 章），或者对锁的使用进行了优化，比如使用粒度极小的锁。因此，并发集合的可伸缩性（`Scalability`）一般要比相应的同步集合高，即使用并发集合的程序相比于使用相应同步集合的程序而言，并发线程数的增加所带来的程序的吞吐率的提升要更加显著¹⁰。而使用同步集合的程序随着并发线程数量的上升，这些同步集合内部所使用的锁的争用所导致的上下文切换开销越来越大，最终有可能使程序的吞吐率一定程度上降低或者恒定到一定的水平。

`ConcurrentLinkedQueue` 是 `Queue` 接口的一个线程安全实现类，它相当于 `LinkedList`（也是 `Queue` 接口的一个实现类）的线程安全版，可以作为 `Collections.synchronizedList`（`new LinkedList()`）的替代品。`ConcurrentLinkedQueue` 内部访问其共享状态变量（如队首指针和队尾指针）的时候并不借助锁，而是使用 CAS 操作来保障线程安全的。因此，`ConcurrentLinkedQueue` 是非阻塞的（`Non-blocking`），其使用不会导致当前线程被暂停，

⁹ 即调用 `Iterator.remove()` 会导致 `UnsupportedOperationException` 异常的抛出。

¹⁰ 当然，在一个处理器上一次只能运行一个线程的情况下，这里有个前提，就是总的并发线程数量不能超过总的处理器数目太多。

因此也就避免了上下文切换的开销。ConcurrentLinkedQueue 所使用的遍历方式是准实时。与 BlockingQueue 的实现类相比, ConcurrentLinkedQueue 更适合于更新操作和遍历操作并发的场景, 比如一个 (或者多个) 线程往/从队列中添加/删除元素, 而另外一个 (或者多个) 线程则对相应队列进行遍历操作。而 BlockingQueue 的实现类 (如 ArrayBlockingQueue) 更适合于多个线程并发更新同一队列的场景, 比如在生产者-消费者模式中生产者线程往队列中添加元素 (产品), 而消费者线程从队列中移除 (消费) 元素。

ConcurrentHashMap 是 Map 接口的一个线程安全实现类, 它相当于 HashMap (也是 Map 接口的一个实现类) 的线程安全版, 可以作为 Hashtable 和 Collections.synchronizedMap (new HashMap()) 的替代品。ConcurrentHashMap 内部使用了粒度极小的锁来保障其线程安全¹¹。ConcurrentHashMap 的读取操作 (如调用 ConcurrentHashMap.get 方法) 基本上不会导致锁的使用。另外, 默认情况下 ConcurrentHashMap 可以支持 16 个并发更新线程, 即这些线程可以在不导致锁 (ConcurrentHashMap 内部使用的锁) 的争用情况下进行并发更新。因此, ConcurrentHashMap 可以支持比较高的并发性, 并且其锁的开销一般比较小。ConcurrentHashMap 中一个构造器支持的 concurrencyLevel 参数可以使我们调整 ConcurrentHashMap 支持的并发更新线程数。当然, 既然这个值是可以调整的, 那么这个值就不会是越大或者越小就越好。这个值越大表示相应的开销越大, 越小表示它越可能导致并发更新时出现锁的争用。因此, concurrencyLevel 的值要调整也必须是根据实际需要来权衡。ConcurrentHashMap 所使用的遍历方式是准实时。第 7 章的一个实战案例会涉及 ConcurrentHashMap 的使用, 参见清单 7-9。

CopyOnWriteArrayList 是 List 接口的一个线程安全实现类, 它相当于 ArrayList (也是 List 接口的一个实现类) 的线程安全版。CopyOnWriteArrayList 内部会维护一个实例变量 array 用于引用一个数组。该数组用于存储列表的各个元素。CopyOnWriteArrayList 的更新操作 (添加、修改和删除) 是通过创建一个新的数组 newArray, 并把老的数组 (array 当前所引用的数组) 的内容复制到 newArray, 然后对 newArray 进行更新并将 array 引用指向 newArray。因此, array 所引用的数组相当于当前 CopyOnWriteArrayList 实例的一个快照, 而对 CopyOnWriteArrayList 的更新操作所导致的对象的复制 (主要是对象引用的复制) 的开销相当于这个快照的间接开销。CopyOnWriteArrayList 所使用的遍历方式就是快照。因此, CopyOnWriteArrayList 适用于遍历操作远比更新操作 (增加、删除和修改) 频繁或者不希望在遍历的时候加锁的场景。而在其他场景下, 我们可能仍然要考虑使用 Collections.synchronizedList(new ArrayList())。

11 详情参见本书第 12 章。

`CopyOnWriteArraySet` 是 `Set` 接口的一个线程安全实现类，它相当于 `HashSet`（也是 `Set` 接口的一个实现类）的线程安全版。`CopyOnWriteArraySet` 内部实现使用了一个 `CopyOnWriteArrayList` 实例，因此 `CopyOnWriteArraySet` 的适用场景与 `CopyOnWriteArrayList` 相似。

6.7 本章小结

本章从面向对象编程的角度出发讲解了实现线程安全的几种常用技术。这些技术的运用通常可以产生具有固有线程安全性的对象，即这些对象本身无须借助锁就可以保障线程安全，从而有利于提高系统的并发性。本章还介绍了同步集合和并发集合。本章知识结构如图 6-5 所示。

Java 运行时空间可分为堆空间、非堆空间以及栈空间。栈空间是线程的私有空间，而堆空间和非堆空间都是线程的共享空间。堆空间用于存储对象以及类的实例变量，它是 Java 虚拟机启动时分配的可以动态扩容的存储空间。非堆空间用于存储类的静态变量以及其他元数据，它是 Java 虚拟机启动时分配的可以动态扩容的存储空间。栈空间用于存储线程所执行的方法的局部变量、返回值等私有数据，它是线程创建时分配的容量固定不可变的存储空间。

无状态对象不包含任何实例变量以及可更新的静态变量，它具有固有的线程安全性。无状态对象的客户端代码在调用该对象的任何方法时都无须加锁，而无状态对象自身的方法实现可能仍然需要借助锁。仅包含静态方法的类并不能取代无状态对象。`Servlet` 类通常需要被设计为无状态对象。

不可变对象也具有固有的线程安全性。严格意义上的不可变对象需要同时满足这几个条件：类本身采用 `final` 修饰，所有字段都是 `final` 字段，在对象初始化过程中 `this` 代表的当前对象没有逸出，引用了状态可变的对象的字段不能直接暴露给其他对象。如果需要将引用了状态可变的对象的字段暴露给其他对象，那么需要在返回该对象前进行防御性复制，或者返回一个不支持 `remove()` 的 `Iterator` 实例。使用不可变对象建模时，系统状态的变化是通过创建新的不可变对象实现的。这种方式可能有利于提高垃圾回收效率，但也可能由于系统状态频繁变更、无状态对象占用较多内存空间等因素增加了垃圾回收的负担。不可变对象的典型应用场景包括：被建模对象的状态变化不频繁、同时对一组相关的数据进行写操作，因此需要保证原子性、使用不可变对象作为安全可靠的 `Map` 键。当被建模对象的状态变更比较频繁时，不可变对象也不见得就不能使用。此时，我们需要综合考虑被建模对象的规模、代码目标运行环境的 Java 虚拟机堆内存容量、系统对吞吐率和响应性的要求这几个因素。

线程特有对象也具有固有的线程安全性。ThreadLocal 是线程访问其线程特有对象的代理。ThreadLocal 也被称为线程局部变量，一个线程可以通过使用不同的线程局部变量来访问不同的线程特有对象实例。多个线程即使是使用同一个线程局部变量，其访问到的对象也是各自的线程特有对象。线程局部变量通常作为一个类的静态字段来使用。为避免线程局部变量的使用导致内存泄漏和伪内存泄漏，我们需要确保在线程特有对象不再被需要的时候将其“删除”（即调用 ThreadLocal.remove()）。线程特有对象的典型应用场景包括：需要使用非线程安全对象，但又不希望因此而引入锁；使用线程安全对象，但希望避免其使用的锁的开销和相关问题；实现方法间的隐式参数传递；实现特定于线程的单例模式。

装饰器模式也能够用于实现线程安全。在使用装饰器模式的情况下，实现同一组功能的对象有非线程安全版和线程安全版两种。这两种对象具有相同的接口，其中非线程安全版对象仅关注功能的实现，而外包装对象（线程安全版）主要关注线程安全的保障。外包装对象在功能方面则是通过委托给相应的非线程安全对象来实现的。Java 并发集合就是使用装饰器模式来保障线程安全的。使用装饰器模式实现线程安全的优点是它支持关注点分离，并有利于降低开发难度和提高代码的可测试性，也有利于提高使用的灵活性。其缺点是并发性不高，并可能导致遍历操作是非线程安全的。

并发集合一般可用于替代同步集合。其内部实现往往借助于 CAS 操作或者细粒度锁。并发集合支持线程安全的遍历操作，即对集合的遍历操作与更新操作是可以由不同线程并发执行的。并发集合实现线程安全的遍历操作有两种方式：快照和准实时。前者无法在遍历过程中反映其他线程对被遍历集合所做的更新，而后者在遍历过程中可能反映其他线程对被遍历集合所做的更新。CopyOnWriteArrayList 相当于 ArrayList 的线程安全版，它适用于遍历操作远比更新操作频繁或者不希望遍历的时候加锁的场景，在其他场景下我们仍然要考虑使用相应的同步集合。CopyOnWriteArraySet 相当于 HashSet 的线程安全版，内部实现是基于 CopyOnWriteArrayList 的。因此，CopyOnWriteArraySet 适用场景与 CopyOnWriteArrayList 类似。ConcurrentLinkedQueue 相当于 LinkedList 的线程安全版，与 BlockingQueue 的实现类相比，ConcurrentLinkedQueue 适用于更新操作和遍历操作并发的场景。BlockingQueue 的实现类更适用于多个线程并发更新同一队列的场景，如生产者—消费者模式中。ConcurrentHashMap 相当于 HashMap 的线程安全版，它能够支持较高的并发性。ConcurrentSkipListMap 相当于 TreeMap 的线程安全版。ConcurrentSkipListSet 相当于 TreeSet 的线程安全版。

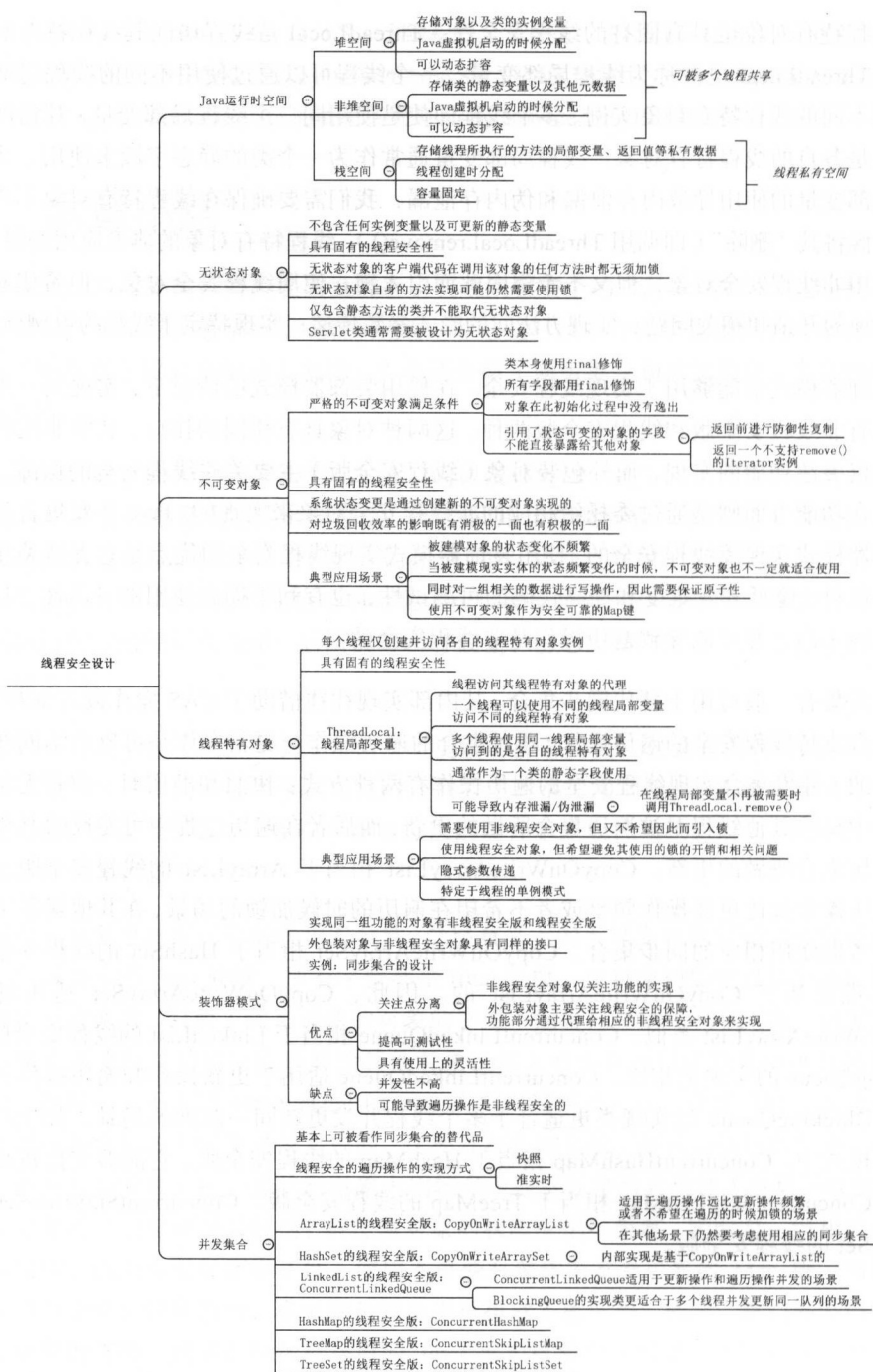


图 6-5 本章知识结构图

线程的活性故障

凡是可能出错的，都会出错！

——墨菲定律

我们已经介绍过线程活性故障这一概念。线程活性故障是由资源稀缺性或者程序自身的问题和缺陷导致线程一直处于非 RUNNABLE 状态，或者线程虽然处于 RUNNABLE 状态但是其要执行的任务却一直无法进展的故障现象。本章我们将从故障成因、影响、检测、规避与恢复这几个方面入手，进一步介绍常见的线程活性故障。

7.1 鹬蚌相争：死锁

死锁是线程的一种常见活性故障。如果两个或者更多的线程因相互等待对方而被永远暂停（线程的生命周期状态为 BLOCKED 或者 WAITING），那么我们就称这些线程产生了死锁（Deadlock）。由于产生死锁的线程的生命周期状态永远是非运行状态，因此这些线程所要执行的任务也永远无法进展。死锁产生的一种典型情形是线程 A 在持有锁 L_1 的情况下申请锁 L_2 ，而线程 B 在持有 L_2 的情况下申请 L_1 ，A 只有在获得并释放 L_2 后才会释放 L_1 ，而 B 只有在获得并释放 L_1 后才会释放 L_2 。也就是说，A 和 B 各自都在持有一个锁（分别为 L_1 和 L_2 ）的情况下去申请对方持有的另外一个锁（分别为 L_2 和 L_1 ），而 A 和 B 释放其持有的锁的前提又都是先获得对方持有的另外一个锁，因此这两个线程最终都无法获得它们申请的另外一个锁，两个线程都处于无限等待的状态，即产生了死锁，如图 7-1 所示。死锁好比鹬蚌相争故事中的情形：鹬啄住蚌的肉，蚌夹住鹬的嘴。鹬对蚌说：“你先放开我的嘴我便不啄你的肉。”而蚌对鹬说：“你先放开我的肉我便不夹你的嘴。”于是最后谁也不放开谁！

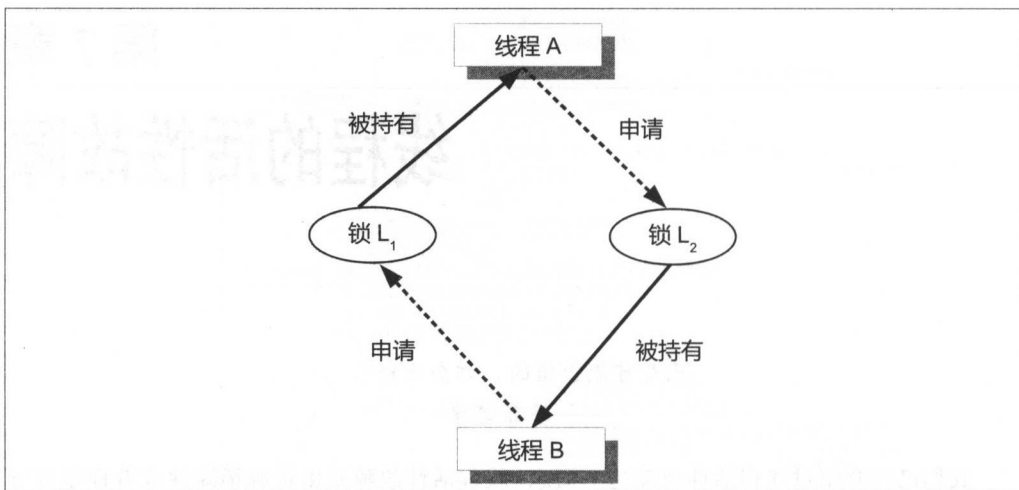


图 7-1 死锁示意图

下面通过一个 Demo 来讲解死锁的检测与规避。

7.1.1 死锁的检测

有关死锁的一个经典问题就是哲学家就餐问题（Dining Philosophers Problem）¹。5 个沉默寡言的哲学家相约去了一家“另类”的中餐馆聚餐。哲学家们围坐在一张大圆桌上，每个座位前面各有一个饭碗，每个饭碗之间都有一根（而不是一双！）筷子（Chopstick），桌子中间有一碗怎么都吃不完的米饭，如图 7-2 所示。每个哲学家（Philosopher）就这么各自对着桌上的米饭坐着，他们要么在思考，要么在吃饭，彼此之间也不交流。吃饭的时候，每个哲学家总是先拿起自己左手边的筷子，再拿起自己右手边的筷子，只有手上持有一双筷子的时候哲学家才能够吃饭。哲学家吃着吃着会放下手中的筷子进行思考，思考完毕之后又接着吃饭，如此这般地在思考与吃饭之间反复。

1 参见：https://en.wikipedia.org/wiki/Dining_philosophers_problem。

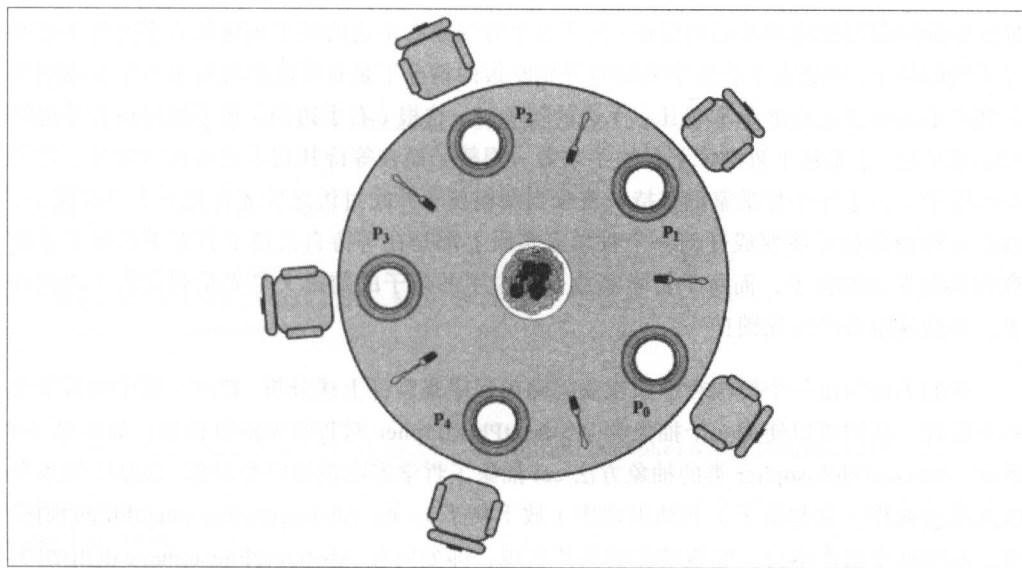


图 7-2 哲学家就餐问题示意图

我们不难写出一个模拟哲学家就餐问题的程序,但是这个程序很可能因为死锁而导致所有哲学家都无法吃饭。为什么呢?我们不妨将这个问题做一下简化:在保持问题的其他约束不变的情况下,我们假设聚餐的哲学家只有两个(Philosopher_1 和 Philosopher_2),他们面对面地坐在一张小方桌上,桌面中央有两根筷子(Chopstick_1 和 Chopstick_2)。经过这样的简化,不难想象在两个哲学家同时试图吃饭的时候很可能出现这样的情形:一个哲学家(哲学家 Philosopher_1)眼明手快拿起了其左手边的筷子,而当他准备拿起其右手边的筷子时,另外一个哲学家(Philosopher_2)也恰好拿起了其(Philosopher_2)左手边的筷子。由于一个哲学家的右手边恰是其相邻的另外一个哲学家的左手边,因此 Philosopher_2 拿起的这根筷子恰是 Philosopher_1 要拿起的那根。此时,每个哲学家都仅拿到一根筷子,而他们都坚持只有拿到对方手上的筷子以便吃饭,才肯放下自己手中的筷子。我们不难看出这种情形正好符合前文提到的死锁的一个典型情形。如果把每个哲学家看作一个线程(哲学家线程),那么由于筷子的数量等于哲学家数量而不是哲学家数量的两倍,因此筷子可以被看作线程间的共享资源。显然筷子是一个独占资源(一根筷子无法同时被两个哲学家使用),因此哲学家线程在访问这些资源的时候就需要加锁。所以,简化后的问题就相当于一个线程(Philosopher_1)在持有一个锁(Chopstick_1 对应的内部锁)的情况下去申请另外一个锁(Chopstick_2 对应的内部锁),而另外一个线程(Philosopher_2)也在持有一个锁(Chopstick_2 对应的内部锁)的情况下去申请另外一个锁(Chopstick_1 对应的内部锁)。可见,简化后的问题是有可能出现死锁的。接下来,我们将简化后的问题推广到原问题,那么不

难想象原问题可能出现类似的情形：由于每个哲学家右手边的筷子正是其右手边哲学家的左手边的筷子，因此在 5 个哲学家同时开始吃饭的情况下是有可能出现每个哲学家刚刚拿起其左手边筷子之后准备拿起其右手边的筷子时，这根（右手边的）筷子恰好被右手边的哲学家拿起，于是每个哲学家都是左手拿着一根筷子而在等待其右手边的哲学家放下其手中的筷子。由于每个哲学家都坚持只有拿到两根筷子并吃过饭之后才肯放下手中的筷子，因此这种情形最终演变成任何一个哲学家实质上都是在等待自己放下其左手的筷子才能拿到其右手边的筷子，而每个哲学家放下其左手的筷子的前提又是先拿到其右手边的筷子，显然这也会产生死锁！

我们不难写出一个模拟哲学家就餐问题的程序来验证上述分析。把该问题中的哲学家看作线程，我们可以使用一个抽象类 `AbstractPhilosopher` 对哲学家进行抽象，如清单 7-1 所示。`AbstractPhilosopher` 类的抽象方法 `eat` 抽象了哲学家吃饭的完整动作，包括吃饭本身以及准备动作（拿起筷子）和结束动作（放下筷子）。而 `AbstractPhilosopher.doEat()` 则模拟了吃饭这个动作本身。吃饭动作的具体实现，即如何在 `AbstractPhilosopher.eat()` 中组织好准备动作、吃饭动作本身以及结束动作这三者之间的关系则由 `AbstractPhilosopher` 类的子类来负责。`AbstractPhilosopher.run()` 则代表哲学家总是在思考与吃饭之间反复。

清单 7-1 表示哲学家的抽象类 `AbstractPhilosopher`

```
/**
 * 对哲学家进行抽象
 *
 * @author Viscent Huang
 */
public abstract class AbstractPhilosopher extends Thread implements Philosopher {
    protected final int id;
    protected final Chopstick left;
    protected final Chopstick right;

    public AbstractPhilosopher(int id, Chopstick left, Chopstick right) {
        super("Philosopher-" + id);
        this.id = id;
        this.left = left;
        this.right = right;
    }

    @Override
    public void run() {
        for (;;) {
            think();
            eat();
        }
    }
}
```

```

}

/*
 * @see io.github.viscent.mtia.ch7.diningphilosophers.Philosopher#eat()
 */
@Override
public abstract void eat();

protected void doEat() {
    Debug.info("%s is eating...%n", this);
    Tools.randomPause(10);
}

/*
 * @see io.github.viscent.mtia.ch7.diningphilosophers.Philosopher#think()
 */
@Override
public void think() {
    Debug.info("%s is thinking...%n", this);
    Tools.randomPause(10);
}

@Override
public String toString() {
    return "Philosopher-" + id;
}
}

```

由于筷子的数量小于哲学家的人数的两倍，因此筷子可被看作哲学家（线程）的共享资源。这里，我们选择用一个非线程安全的类 Chopstick 表示筷子，如清单 7-2 所示。

清单 7-2 筷子模型 Chopstick 类

```

/**
 * 筷子
 *
 * @author Viscent Huang
 */
public class Chopstick {
    public final int id;
    private Status status = Status.PUT_DOWN;

    public Chopstick(int id) {
        super();
        this.id = id;
    }

    public void pickUp() {

```

```

        status = Status.PICKED_UP;
    }

    public void putDown() {
        status = Status.PUT_DOWN;
    }
    // 完整代码见本书配套下载资源
}

```

接下来我们需要创建一个 `AbstractPhilosopher` 类的子类来实现吃饭动作，如清单 7-3 所示。由于 `Chopstick` 是一个非线性安全对象，因此 `AbstractPhilosopher.eat()` 在调用 `Chopstick.pickUp()/putDown()` 来模拟拿起筷子/放下筷子的时候需要加锁。再加上哲学家总是先拿起其左手边的筷子，然后才拿起其右手边的筷子，因此我们使用了一个嵌套的同步块并在相应的临界区中调用 `Chopstick.pickUp()`。由于哲学家只有在拿到两根筷子的情况下才能够吃饭，因此 `AbstractPhilosopher.doEat()` 调用必须放在嵌套同步块的内层同步块的临界区中（这样同时持有两根筷子这个条件才能成立）。

清单 7-3 可能产生死锁的 `AbstractPhilosopher` 子类 `DeadlockingPhilosopher`

```

/**
 * 能导致死锁的哲学家模型
 *
 * @author Viscent Huang
 */
public class DeadlockingPhilosopher extends AbstractPhilosopher {
    public DeadlockingPhilosopher(int id, Chopstick left, Chopstick right) {
        super(id, left, right);
    }

    @Override
    public void eat() {
        synchronized (left) {
            Debug.info("%s is picking up %s on his left...\n", this, left);
            left.pickUp(); // 拿起左手边的筷子
            synchronized (right) {
                Debug.info("%s is picking up %s on his right...\n", this, right);
                right.pickUp(); // 拿起右手边的筷子
                doEat(); // 同时拿起两根筷子的时候才能够吃饭
                right.putDown();
            }
            left.putDown();
        }
    }
}

```

接着，我们便可以写出哲学家就餐问题的模拟程序本身，如清单 7-4 所示。

清单 7-4 哲学家就餐问题模拟程序

```
public class DiningPhilosopherProblem {

    public static void main(String[], args) throws Exception {
        int numOfPhilosophers;
        numOfPhilosophers = args.length > 0 ? Integer.valueOf(args[0]) : 2;
        // 创建筷子
        Chopstick[] chopsticks = new Chopstick[numOfPhilosophers];
        for (int i = 0; i < numOfPhilosophers; i++) {
            chopsticks[i] = new Chopstick(i);
        }

        String philosopherImplClassName = System.getProperty("x.philo.impl");
        if (null == philosopherImplClassName) {
            philosopherImplClassName = "DeadlockingPhilosopher";
        }

        for (int i = 0; i < numOfPhilosophers; i++) {
            // 创建哲学家
            createPhilosopher(philosopherImplClassName, i, chopsticks);
        }
    }

    private static void createPhilosopher(String philosopherImplClassName,
        int id, Chopstick[] chopsticks) throws Exception {

        int numOfPhilosophers = chopsticks.length;
        @SuppressWarnings("unchecked")
        Class<Philosopher> philosopherClass = (Class<Philosopher>) Class
            .forName(DiningPhilosopherProblem.class.getPackage().getName() + "."
                + philosopherImplClassName);
        Constructor<Philosopher> constructor = philosopherClass.getConstructor(
            int.class, Chopstick.class, Chopstick.class);
        Philosopher philosopher = constructor.newInstance(id, chopsticks[id],
            chopsticks[(id + 1)
                % numOfPhilosophers]);
        philosopher.start();
    }
}
```

在不指定任何参数的情况下运行上述程序，不用多久我们便可以发现程序“卡住”了而没有任何新的输出。此时检查该程序的线程转储（Thread Dump）可以发现该程序出现

了死锁²：

Found one Java-level deadlock:

```
=====
"Philosopher-1":
  waiting to lock monitor 0x00007f7bc80062c8 (object 0x00000000d739e2f0, a mtia.
  Chopstick),
  which is held by "Philosopher-0"
"Philosopher-0":
  waiting to lock monitor 0x00007f7bc80022d8 (object 0x00000000d73a31d0, a mtia.
  Chopstick),
  which is held by "Philosopher-1"
```

Java stack information for the threads listed above:

```
=====
"Philosopher-1":
  at mtia.DeadlockingPhilosopher.eat(DeadlockingPhilosopher.java:16)
  - waiting to lock <0x00000000d739e2f0> (a mtia.Chopstick)
  - locked <0x00000000d73a31d0> (a mtia.Chopstick)
  at mtia.AbstractPhilosopher.run(AbstractPhilosopher.java:21)
"Philosopher-0":
  at mtia.DeadlockingPhilosopher.eat(DeadlockingPhilosopher.java:16)
  - waiting to lock <0x00000000d73a31d0> (a mtia.Chopstick)
  - locked <0x00000000d739e2f0> (a mtia.Chopstick)
  at mtia.AbstractPhilosopher.run(AbstractPhilosopher.java:21)
```

Found 1 deadlock.

由于本程序默认的哲学家数量为 2（相当于模拟简化后的问题），因此这个死锁仅涉及两个线程，它们的名称分别为 Philosopher-0 和 Philosopher-1。这两个线程都是在持有一个对方所申请的锁的情况下去申请对方持有的锁（如表 7-1 所示），从而产生了死锁。

表 7-1 两个哲学家线程互相申请对方持有的锁

	申请的锁	持有的锁
Philosopher-1	0x00007f7bc80062c8	0x00007f7bc80022d8
Philosopher-0	0x00007f7bc80022d8	0x00007f7bc80062c8

在清单 7-3 中我们使用内部锁来实现 AbstractPhilosopher.eat()。这个方法也能够用显式锁实现，如清单 7-5（BuggyLckBasedPhilosopher 类）所示。BuggyLckBasedPhilosopher 为每个 Chopstick 实例创建一个唯一与之对应（而不重复）的 ReentrantLock 实例，并在调

2 有关线程转储的信息参见本书第 1 章。为了便于排版，笔者把线程转储中的真实包名 io.github.viscent.mtia.ch7.diningphilosophers 替换成 mtia。

用 `Chopstick.pickUp()/putDown()` 时使用相应的 `ReentrantLock` 实例进行加锁。

清单 7-5 可能产生死锁的基于显式锁的 `AbstractPhilosopher` 子类 `BuggyLckBasedPhilosopher`

```
public class BuggyLckBasedPhilosopher extends AbstractPhilosopher {
    /**
     * 为确保每个 Chopstick 实例有且仅有一个显式锁（而不重复创建）与之对应，<br>
     * 这里的 map 必须采用 static 修饰！
     */
    protected final static ConcurrentMap<Chopstick, ReentrantLock> LOCK_MAP;
    static {
        LOCK_MAP = new ConcurrentHashMap<Chopstick, ReentrantLock>();
    }

    public BuggyLckBasedPhilosopher(int id, Chopstick left, Chopstick right) {
        super(id, left, right);
        // 每个筷子对应一个（唯一）锁实例
        LOCK_MAP.putIfAbsent(left, new ReentrantLock());
        LOCK_MAP.putIfAbsent(right, new ReentrantLock());
    }

    @Override
    public void eat() {
        // 先后拿起左边和右手边的筷子
        if (pickUpChopstick(left) && pickUpChopstick(right)) {
            // 同时拿起两根筷子的时候才能够吃饭
            try {
                doEat();
            } finally {
                // 放下筷子
                putDownChopsticks(right, left);
            }
        }
    }

    protected boolean pickUpChopstick(Chopstick chopstick) {
        final ReentrantLock lock = LOCK_MAP.get(chopstick);
        lock.lock();
        try {
            Debug.info("%s is picking up %s on his %s...%n",
                this, chopstick, chopstick == left ? "left" : "right");
            chopstick.pickUp();
        } catch (Exception e) {
            // 不大可能走到这里
            e.printStackTrace();
            lock.unlock();
            return false;
        }
    }
}
```



```

        return true;
    }
    private void putDownChopsticks(Chopstick chopstick1, Chopstick chopstick2) {
        try {
            putDownChopstick(chopstick1);
        } finally {
            putDownChopstick(chopstick2);
        }
    }
    protected void putDownChopstick(Chopstick chopstick) {
        final ReentrantLock lock = LOCK_MAP.get(chopstick);
        try {
            Debug.info("%s is putting down %s on his %s...%n",
                this, chopstick, chopstick == left ? "left" : "right");
            chopstick.putDown();
        } finally {
            lock.unlock();
        }
    }
}

```

使用显式锁实现的 `AbstractPhilosopher.eat()` 也同样可能导致死锁！使用如下命令将 `BuggyLckBasedPhilosopher` 指定为哲学家实现类来运行本 Demo：

```

java -Dx.philo.impl=BuggyLckBasedPhilosopher
io.github.viscent.mtia.ch7.diningphilosophers.DiningPhilosopherProblem

```

通过获取并查看此时的线程转储我们仍然可以发现死锁³：

Found one Java-level deadlock:

=====

"Philosopher-1":

waiting for ownable synchronizer 0x00000000d716dd68, (a `jl.ReentrantLock$NonfairSync`), which is held by "Philosopher-0"

"Philosopher-0":

waiting for ownable synchronizer 0x00000000d716de08, (a `jl.ReentrantLock$NonfairSync`), which is held by "Philosopher-1"

Java stack information for the threads listed above:

=====

"Philosopher-1":

```

    at sun.misc.Unsafe.park(Native Method)
    - parking to wait for <0x00000000d716dd68> (a jl.ReentrantLock$NonfairSync)
    at jl.LockSupport.park(LockSupport.java:175)

```

3 为了便于排版，笔者把该线程转储中的真实包名 `java.util.concurrent.locks` 替换为 `jl`，把 `io.github.viscent.mtia.ch7.diningphilosophers` 替换成 `mtia`。

```

    at java.lang.AbstractQueuedSynchronizer.parkAndCheckInterrupt (AbstractQueuedSync
hronizer.java:836)
    at java.lang.AbstractQueuedSynchronizer.acquireQueued (AbstractQueuedSynchronizer.
java:870)
    at java.lang.AbstractQueuedSynchronizer.acquire (AbstractQueuedSynchronizer.java:
1199)
    at java.lang.ReentrantLock$NonfairSync.lock (ReentrantLock.java:209)
    at java.lang.ReentrantLock.lock (ReentrantLock.java:285)
    at mtia.BuggyLckBasedPhilosopher.pickUpChopstick (BuggyLckBasedPhilosopher.
java:37)
    at mtia.BuggyLckBasedPhilosopher.eat (BuggyLckBasedPhilosopher.java:27)
    at mtia.AbstractPhilosopher.run (AbstractPhilosopher.java:26)

"Philosopher-0":
    at sun.misc.Unsafe.park (Native Method)
    - parking to wait for <0x00000000d716de08> (a java.lang.ReentrantLock$NonfairSync)
    at java.lang.LockSupport.park (LockSupport.java:175)
    at java.lang.AbstractQueuedSynchronizer.parkAndCheckInterrupt (AbstractQueuedSync
hronizer.java:836)
    at java.lang.AbstractQueuedSynchronizer.acquireQueued (AbstractQueuedSynchronizer.
java:870)
    at java.lang.AbstractQueuedSynchronizer.acquire (AbstractQueuedSynchronizer.java:
1199)
    at java.lang.ReentrantLock$NonfairSync.lock (ReentrantLock.java:209)
    at java.lang.ReentrantLock.lock (ReentrantLock.java:285)
    at mtia.BuggyLckBasedPhilosopher.pickUpChopstick (BuggyLckBasedPhilosopher.
java:37)
    at mtia.BuggyLckBasedPhilosopher.eat (BuggyLckBasedPhilosopher.java:27)
    at mtia.AbstractPhilosopher.run (AbstractPhilosopher.java:26)

Found 1 deadlock.

```

7.1.2 死锁产生的条件与规避

哲学家就餐问题反映了产生死锁的必要条件，线程一旦产生死锁，那么这些线程及相关的资源将满足如下全部条件⁴。

- 资源互斥（Mutual Exclusion）。涉及的资源必须是独占的，即每个资源一次只能被一个线程使用。例如，哲学家就餐问题中的筷子（或者使用筷子时所需持有的锁）可被看作独占的资源。

⁴ 参见：https://en.wikipedia.org/wiki/Deadlock#Necessary_conditions。

- 资源不可抢夺（No Preemption）。涉及的资源只能够被其持有者（线程）主动释放，而无法被资源的持有者和申请者之外的第三线程所抢夺（被动释放）。例如，哲学家就餐问题中的筷子只能由持有该筷子的哲学家（线程）主动放下（释放）。
- 占用并等待资源（Hold and Wait）。涉及的线程当前至少持有一个资源（资源 A）并申请其他资源（资源 B），而这些资源（资源 B）恰好被其他线程持有。在这个资源等待的过程中，线程并不释放其已经持有的资源。例如，哲学家就餐问题中一个哲学家（线程）左手拿着筷子（资源 A）而等待其右手边的筷子（资源 B），这根筷子恰好被其右手边的哲学家（线程）拿起（持有）。并且，等待其他哲学家手上的筷子的哲学家并不放下自己手中的筷子。
- 循环等待资源（Circular Wait）。涉及的线程必须在等待别的线程持有的资源，而这些线程又反过来在等待第 1 个线程所持有的资源。比如有一组线程 $\{T_1, T_2, \dots, T_N\}$ 以及一组资源 $\{R_1, R_2, \dots, R_N\}$ ， T_1 在等待 R_2 而 R_2 被 T_2 持有， T_2 在等待 R_3 而 R_3 被 T_3 持有，……， T_N 在等待 R_1 而 R_1 被 T_1 持有，这些线程就满足了循环等待资源这个条件。例如，哲学家就餐问题中第 1 个哲学家（线程）在等待第 2 个哲学家左手持有的筷子（资源），第 2 个哲学家（线程）在等待第 3 个哲学家左手持有的筷子（资源），……，第 5 个哲学家在等待第 1 个哲学家左手持有的筷子。

这些条件是死锁产生的必要条件而非充分条件，也就是说只要产生了死锁，那么上面这些条件一定同时成立，但是上述条件即使同时成立也不一定就能产生死锁。因此，死锁和其他的多线程相关的问题（比如可见性问题）类似，它并不是必然出现的！上述几个条件并非完全独立，其中“循环等待资源”就可能蕴含了“占用并等待资源”，而“占用并等待资源”可能是“循环等待资源”的基础，但却不一定意味着“循环等待资源”。

我们可以把锁看作一种资源，这种资源正好符合“资源互斥”和“资源不可抢夺”的要求。那么，可能产生死锁的代码特征就是在持有一个锁的情况下去申请另外一个锁，这通常意味着锁的嵌套，如图 7-3 所示。

内部锁	显式锁
<pre> public void deadlockProne() { synchronized (lockA) { // ... synchronized (lockB) { // ... } } } </pre>	<pre> public void deadlockProne() { lockA.lock(); try { // ... lockB.lock(); try { // ... } finally { lockB.unlock(); } } finally { lockA.unlock(); } } } </pre>

图 7-3 死锁的代码特征

一个线程在已经持有一个锁的情况下再次申请这个锁（比如，一个类的一个同步方法调用该类的另外一个同步方法）并不会导致死锁，这是因为 Java 中的锁（包括内部锁和显式锁）都是可重入的（Reentrant），这种情形下线程再次申请这个锁是可以成功的⁵。

弄清楚死锁产生的必要条件也就不难想到规避死锁的方法——我们只要消除死锁产生的任意一个必要条件就可以规避死锁了。由于锁具有排他性并且锁只能够由其持有线程主动释放，因此由锁导致的死锁只能够从消除“占用并等待资源”和消除“循环等待资源”这两个方向入手。相应地，下面我们介绍基于这两个思路规避死锁的方法。

粗锁法（Coarsen-grained Lock）——使用粗粒度的锁代替多个锁。从消除“占用并等待资源”出发我们不难想到的一种方法就是，采用一个粒度较粗的锁来替代原先的多个粒度较细的锁，这样涉及的线程都只需要申请一个锁从而避免了死锁。按照这个思路，我们可以编写一个能够规避死锁的 `AbstractPhilosopher` 实现类 `GlobalLckBasedPhilosopher`，如清单 7-6 所示。`GlobalLckBasedPhilosopher.eat()` 会使用一个静态变量 `GLOBAL_LOCK` 作为锁。这样，所有哲学家线程（`GlobalLckBasedPhilosopher` 的实例）在拿起筷子前都必须持有 `GLOBAL_LOCK` 对应的内部锁。此时，由于每个哲学家线程仅需要申请一个锁就可以吃饭，因此死锁产生的必要条件“占用并等待资源”和“循环等待资源”就都不成立了，从而避免了死锁。

清单 7-6 使用粗粒度的锁规避死锁

```
public class GlobalLckBasedPhilosopher extends AbstractPhilosopher {
```

⁵ 实际上也是有限制的，对于显式锁，一个线程进行这样的锁申请最多只能进行 2 147 483 647 次。

```

// GLOBAL_LOCK 必须使用 static 修饰
private final static Object GLOBAL_LOCK = new Object();
public GlobalLckBasedPhilosopher(int id, Chopstick left,
    Chopstick right) {
    super(id, left, right);
}

@Override
public void eat() {
    synchronized (GLOBAL_LOCK) {
        Debug.info("%s is picking up %s on his left...%n", this, left);
        left.pickUp();
        Debug.info("%s is picking up %s on his right...%n", this, right);
        right.pickUp();
        doEat();
        right.putDown();
        left.putDown();
    }
} // eat 方法结束
}

```

粗锁法的缺点是它明显地降低了并发性并可能导致资源浪费。例如，GlobalLckBasedPhilosopher.eat()采用粗锁法的结果是一次只能有一个哲学家能够吃饭，一个哲学家在吃饭的时候其他哲学家只能在思考或者等待筷子！而实际上，一个哲学家在吃饭的时候仅占用了两根筷子，剩下的三根筷子其实还够供另外一个哲学家使用！因此，粗锁法的适用范围比较有限。

锁排序法（Lock Ordering）——相关线程使用全局统一的顺序申请锁。假设有多个线程需要申请资源（锁） $\{Lock_1, Lock_2, \dots, Lock_N\}$ ，那么我们只需要让这些线程依照一个全局（相对于使用这种资源的所有线程而言）统一的顺序去申请这些资源，就可以消除“循环等待资源”这个条件，从而规避死锁。例如，在哲学家就餐问题中每个哲学家都是依照“先拿起左手边的筷子，再拿起右手边的筷子”这种局部顺序来拿筷子的。之所以称这种顺序为“局部”，是因为一个哲学家右手边的筷子恰恰是另外一个哲学家左手边的筷子。因此，从全局的角度来看这种拿筷子的顺序实际上是各个线程各自为政使用不同的顺序，从而使“循环等待资源”得以成立。为了消除“循环等待资源”这个死锁产生的必要条件，我们可以让所有的哲学家（线程）使用全局统一的顺序去拿起两根筷子，比如先拿编号（id）值较小的，再拿编号值较大的筷子。这种方法实际上是对资源（筷子或者访问筷子所需的锁）进行排序。一般地，我们可以使用对象的身份 hashCode（Identity Hash Code，即 System.identityHashCode(Object)的返回值）来作为资源的排序依据。依照这个思路，我们可以编写能够规避死锁的 AbstractPhilosopher 实现类 FixedPhilosopher，如清单 7-7 所示。

清单 7-7 使用锁排序规避死锁

```

public class FixedPhilosopher extends AbstractPhilosopher {
    private final Chopstick one;
    private final Chopstick theOther;

    public FixedPhilosopher(int id, Chopstick left, Chopstick right) {
        super(id, left, right);
        // 对资源（锁）进行排序
        int leftHash = System.identityHashCode(left);
        int rightHash = System.identityHashCode(right);
        if (leftHash < rightHash) {
            one = left;
            theOther = right;
        } else if (leftHash > rightHash) {
            one = right;
            theOther = left;
        } else {
            // 两个对象的 identityHashCode 值相等是可能的，尽管这个概率很小
            one = null;
            theOther = null;
        }
    }

    @Override
    public void eat() {
        if (null != one) {
            synchronized (one) {
                Debug.info("%s is picking up %s on his %s...\n", this, one,
                    one == left ? "left" : "right");
                one.pickUp();
            }
            synchronized (theOther) {
                Debug.info("%s is picking up %s on his %s...\n", this,
                    theOther, theOther == left ? "left" : "right");
                theOther.pickUp();
                doEat();
                theOther.putDown();
            }
            one.putDown();
        } else {
            // 退化为使用粗锁法
            synchronized (FixedPhilosopher.class) {
                Debug.info("%s is picking up %s on his left...\n", this, left);
                left.pickUp();
                Debug.info("%s is picking up %s on his right...\n", this, right);
                right.pickUp();
                doEat();
            }
        }
    }
}

```

```

        right.putDown();

        left.putDown();
    }
} // if 语句结束
} // eat 方法结束
}

```

在 `FixedPhilosopher` 类中，我们先在该类的构造器中根据 `Chopstick` 实例的身份 `hashCode` 对左手边的筷子和右手边的筷子进行排序，排序的结果记为一根筷子（`one`）和另外一根筷子（`theOther`）。在 `eat` 方法中，我们在调用 `Chopstick.pickUp()/putDown()` 的时候分别用 `one` 和 `theOther` 去替代 `left`（左手边筷子）和 `right`（右手边筷子）进行加锁。这样就确保每个哲学家线程都是使用全局统一的顺序去申请资源（`Chopstick` 对应的内部锁），从而消除了“循环等待资源”这个条件而规避了死锁。由于使用不同的对象调用 `System.identityHashCode(Object)` 仍然可能返回相同的身份 `hashCode`（尽管这种可能性极低），因此在 `eat` 方法中我们仍然考虑到这些情形（即 `left` 和 `right` 对应的身份 `hashCode` 相同）。此时，我们转而使用粗锁法——使用 `FixedPhilosopher` 类对象本身（Java 平台中的类本身也是一种对象）作为全局锁对 `Chopstick.pickUp()/putDown()` 调用进行加锁。

规避死锁的第 3 种方法是使用 `ReentrantLock.tryLock(long,TimeUnit)` 申请锁。`ReentrantLock.tryLock(long,TimeUnit)` 允许我们为锁申请这个操作指定一个超时时间。在超时时间内，如果相应的锁申请成功，那么该方法返回 `true`；如果在 `tryLock(long,TimeUnit)` 执行的那一刻相应的锁正被其他线程持有，那么该方法会使当前线程暂停，直到这个锁被申请成功（此时该方法返回 `true`）或者等待时间超过指定的超时时间（此时该方法返回 `false`）。因此，使用 `tryLock(long,TimeUnit)` 来申请锁可以避免一个线程无限制地等待另外一个线程持有的资源，从而最终能够消除死锁产生的必要条件中的“占用并等待资源”。使用 `tryLock(long,TimeUnit)` 我们可以编写能够规避死锁的 `AbstractPhilosopher` 实现类 `FixedLockBasedPhilosopher`，如清单 7-8 所示。

清单 7-8 使用 `tryLock(long,TimeUnit)` 规避死锁

```

public class FixedLockBasedPhilosopher extends
    BuggyLckBasedPhilosopher {

    @Override
    protected boolean pickUpChopstick(Chopstick chopstick) {
        final ReentrantLock lock = LOCK_MAP.get(chopstick);
        boolean pickedUp = false;
        boolean lockAcquired = false;
        try {
            lockAcquired = lock.tryLock(50, TimeUnit.MILLISECONDS);

```

```

    if (!lockAcquired) {
        // 锁申请失败
        Debug.info("%s is trying to pick up %s on his %s,"
            + "but it is held by other philosopher ...%n",
            this, chopstick, chopstick == left ? "left" : "right");
        return false;
    }
} catch (InterruptedException e) {
    // 若当前线程已经拿起另外一根筷子, 则使其放下
    Chopstick theOtherChopstick = chopstick == left ? right : left;
    if (LOCK_MAP.get(theOtherChopstick).isHeldByCurrentThread()) {
        theOtherChopstick.putDown();
        LOCK_MAP.get(theOtherChopstick).unlock();
    }
    return false;
}

try {
    Debug.info("%s is picking up %s on his %s...%n",
        this, chopstick, chopstick == left ? "left" : "right");
    chopstick.pickUp();
    pickedUp = true;
} catch (Exception e) {
    // 不大可能走到这里
    if (lockAcquired) {
        lock.unlock();
    }
    pickedUp = false;
    e.printStackTrace();
}
return pickedUp;
}
}

```

FixedLockBasedPhilosopher 类覆盖了其父类的 pickUpChopstick 方法（实现拿起指定的筷子的功能）。这个 pickUpChopstick 方法会调用指定筷子（chopstick）对应的 ReentrantLock 实例的 tryLock(long,TimeUnit)来申请相应的锁。这里，我们指定一个哲学家（即 pickUpChopstick 方法的执行线程）在等待其他哲学家放下其手中的（一根）筷子时最多只等待 50 毫秒，即避免了无限制的等待造成的“占用并等待资源”。

事实上，我们接触到的能够导致死锁的代码可能并不直接具备图 7-3 所示的特征（持有一个锁并申请另外一个锁的特征），更为常见的情况可能是一个方法在持有一个锁的情况下调用一个外部方法（Alien Method，参见 3.11 节）。设类 ClassA 有两个同步方法 syncOperationA 和 syncOperationB，类 ClassB 也有两个同步方法 syncOperationC 和 syncOperationD，

syncOperationA 会调用 syncOperationC，syncOperationD 会调用 syncOperationB。这里，syncOperationA 和 syncOperationD 这两个方法虽然不直接具备图 7-3 所示的特征，但是由于它们调用了外部方法，而这个方法是一个同步方法，因此这两个方法实际上具备了图 7-3 所示的特征。当一个线程在执行 ClassA.syncOperationA()时，另外一个线程正在执行 ClassB.syncOperationD()，那么这两个线程就有可能产生死锁。一般地，一个方法在持有一个锁的情况下调用一个外部方法，而外部方法往往不在我们（开发人员）的控制范围之内，其自身可能不会申请另外一个锁，也可能会申请另外一个锁。因此，在持有一个锁的情况下调用一个外部方法的代码很可能会间接具备图 7-3 所示的特征，从而导致死锁。这种情形导致死锁在一些开源软件甚至于 Java 标准库本身都曾出现过⁶。

下面我们通过一个实战案例来讲解在持有一个锁的情况下调用外部方法导致的死锁。

某系统的实际配置数据存储在数据库之中，该系统有一个配置管理模块，其核心功能是为业务模块提供获取以及缓存系统配置数据的功能、为系统管理模块提供动态更新系统配置数据的功能。该模块的主要类如表 7-2 所示。

表 7-2 某系统配置管理模块的主要类

类	介 绍	源 码
Configuration	配置实体，代表该系统的配置数据。每个配置实体可以包含若干配置条目。每个配置条目是一个从“属性名”到“属性值”的关联	清单 7-11
ConfigurationHelper	配置助手。业务模块通过调用该类的相应方法访问系统的配置实体。该类还能够缓存配置实体	清单 7-10
ConfigurationManager	配置管理器。系统管理模块在更新完系统的配置数据后通过该类将这种更新“通知”到 ConfigurationHelper 以及需要的业务模块对象	清单 7-9

由清单 7-9 可见，ConfigurationManager.update 方法在持有一个锁（ConfigurationManager 当前实例对应的内部锁）的情况下调用了一个外部方法——ConfigEventListener.onConfigUpdated 方法。而 ConfigurationHelper（见清单 7-10）作为 ConfigEventListener 接口的一个实现类，其 onConfigUpdated 方法内部又申请另外一个锁（ConfigurationHelper 当前实例对应的内部锁）。可见，ConfigurationManager.update 方法间接具备了图 7-3 所示的代码特征。另一方面，我们不难发现 ConfigurationHelper.getConfig 方法事实上也具备图 7-3 所示的代码特征：ConfigurationHelper.getConfig 方法可能在持有一个锁（ConfigurationHelper 当前实例对应的内部锁）的情况下调用外部方法——

6 参见 Java 编号为 6927486 的 Bug：http://bugs.java.com/view_bug.do?bug_id=6927486。

`ConfigurationManager.load` 方法,而这个方法本身会申请另外一个锁(`ConfigurationManager` 当前实例对应的内部锁)。因此,如清单 7-12 所示,假如一个业务线程执行 `ConfigurationHelper.getConfig` 方法来获取一个配置实体的时候,另外一个线程(配置管理线程)恰好更新了系统的配置数据,该线程通过执行 `ConfigurationManager.update` 方法将这种更新“通知”给 `ConfigurationHelper` 以及可能的业务模块对象,那么这两个线程就可能产生死锁。

清单 7-9 `ConfigurationManager` 源码

```
/**
 * 配置管理器
 * 该类可能导致死锁!
 * @author Viscent Huang
 */
public enum ConfigurationManager {
    INSTANCE;

    protected final Set<ConfigEventListener> listeners = new
        HashSet<ConfigEventListener>();

    public Configuration load(String name) {
        Configuration cfg = loadConfigurationFromDB(name);
        synchronized (this) {
            for (ConfigEventListener listener : listeners) {
                listener.onConfigLoaded(cfg);
            }
        }
        return cfg;
    }
    // 从数据库加载配置实体(数据)
    private Configuration loadConfigurationFromDB(String name) {
        // ...
    }

    public synchronized void registerListener(ConfigEventListener listener) {
        listeners.add(listener);
    }

    public synchronized void removeListener(ConfigEventListener listener) {
        listeners.remove(listener);
    }

    public synchronized void update(String name, int newVersion,
        Map<String, String> properties) {
        for (ConfigEventListener listener : listeners) {
            // 这个外部方法调用可能导致死锁!
        }
    }
}
```

```

        listener.onConfigUpdated(name, newVersion, properties);
    }
}

```

清单 7-10 ConfigurationHelper 源码

```

/**
 * 配置助手
 * 该类可能导致死锁!
 * @author Viscent Huang
 */
public enum ConfigurationHelper implements ConfigEventListener {
    INSTANCE;

    final ConfigurationManager configManager;
    final ConcurrentMap<String, Configuration> cachedConfig;

    private ConfigurationHelper() {
        configManager = ConfigurationManager.INSTANCE;
        cachedConfig = new ConcurrentHashMap<String, Configuration>();
    }

    public Configuration getConfig(String name) {
        Configuration cfg;
        cfg = getCachedConfig(name);
        if (null == cfg) {
            synchronized (this) {
                cfg = getCachedConfig(name);
                if (null == cfg) {
                    cfg = configManager.load(name);
                    cachedConfig.put(name, cfg);
                }
            }
        }
        return cfg;
    }

    public Configuration getCachedConfig(String name) {
        return cachedConfig.get(name);
    }

    public ConfigurationHelper init() {
        configManager.registerListener(this);
        return this;
    }

    @Override
    public void onConfigLoaded(Configuration cfg) {

```

```

        cachedConfig.putIfAbsent(cfg.getName(), cfg);
    }

    @Override
    public void onConfigUpdated(String name, int newVersion,
        Map<String, String> properties) {
        Configuration cachedConfig = getCacheConfig(name);
        // 更新内容和版本这两个操作必须是原子操作
        synchronized (this) {
            if (null != cachedConfig) {
                cachedConfig.update(properties);
                cachedConfig.setVersion(newVersion);
            }
        }
    }
}

```

清单 7-11 Configuration 源码

```

/**
 * 配置实体
 *
 * @author Viscent Huang
 */
// 非线性安全
public class Configuration {
    /**
     * 配置名称
     */
    private final String name;
    /**
     * 配置当前版本号
     */
    private volatile int version;
    /**
     * 存储配置项的 Map。每个配置项是一个“属性名”->“属性值”的关联
     */
    private volatile Map<String, String> configItemMap;

    public Configuration(String name, int version) {
        this.name = name;
        this.version = version;
        configItemMap = new HashMap<String, String>();
    }

    public void setProperty(String key, String value) {
        configItemMap.put(key, value);
    }
}

```

```
public String getProperty(String key) {
    return configItemMap.get(key);
}

public void update(Map<String, String> properties) {
    configItemMap = properties;
}

public void setVersion(int version) {
    this.version = version;
}

// ...
}
```

清单 7-12 外部方法调用导致的死锁 Demo

```
public class CaseRunner7_1 {

    final static ConfigurationHelper configHelper =
        ConfigurationHelper.INSTANCE.init();

    public static void main(String[] args) throws InterruptedException {
        // 模拟业务线程读取配置实体
        Thread trxThread = new Thread(new Runnable() {

            @Override
            public void run() {
                Configuration cfg = configHelper.getConfig("serverInfo");
                String url = cfg.getProperty("url");
                process(url);
            }

            private void process(String url) {
                Debug.info("processing %s", url);
                // ...
            }

        });

        // 模拟系统管理线程更新配置数据
        Thread updateThread = new Thread(new Runnable() {

            @Override
            public void run() {
                // 模拟实际操作所需的时间
                Tools.randomPause(40);
            }
        });
    }
}
```

```

    Map<String, String> props = new HashMap<String, String>();
    props.put("property1", "value1");
    props.put("property2", "value2");
    props.put("property3", "value3");
    ConfigurationManager.INSTANCE.update("anotherConfig", 6, props);
}

});

// 启动并等待指定的线程终止
Tools.startAndWaitTerminated(trxThread, updateThread);
}
}

```

本案例中出现的死锁可以使用“开放调用”来规避。所谓开放调用（Open Call）就是一个方法在调用外部方法（Alien Method，包括其他类的方法以及当前类的可覆盖方法）的时候不持有任何锁。显然，开放调用能够消除死锁产生的必要条件中的“持有并等待资源”。既然通过上面的分析我们已经锁定了本案例中导致死锁的“罪魁祸首”——ConfigurationManager.update 方法（参见清单 7-9）以及 ConfigurationHelper.getConfig 方法（参见清单 7-10），那么我们只需要将这两个方法对外部方法的调用改为开放调用即可。考虑到将 ConfigurationHelper.getConfig 方法改造为开放调用比较困难，我们不妨从 ConfigurationManager 入手——将 ConfigurationManager 的实例变量 listeners 改用线程安全的 Set 接口实现类 CopyOnWriteArraySet（参见第 6 章），这种改造使得我们可以将 ConfigurationManager 的几个方法，包括 update 方法和 load 方法改为无须申请锁的方法，如清单 7-13 所示。改造后的 ConfigurationManager.update 方法对外部方法 onConfigUpdated 的调用已经是开放调用（类似地，load 方法对外部方法 onConfigLoaded 的调用也是开放调用）。尽管改造之后 ConfigurationHelper.getConfig 方法对 ConfigurationManager.load 方法的调用仍然不是开放调用，但是由于 ConfigurationManager 中所有对 ConfigurationHelper 的方法调用都不持有锁，因此死锁产生的必要条件中的“循环等待”就不会成立，由此我们还是规避了死锁。

清单 7-13 使用开放调用改造 ConfigurationManager

```

public enum ConfigurationManagerV2 {
    INSTANCE;
    protected final Set<ConfigEventListener> listeners;
    {
        listeners = new CopyOnWriteArraySet<ConfigEventListener>();
    }
    // 省略未改动过的代码……
}

```

```

public Configuration load(String name) {
    Configuration cfg = loadConfigurationFromDB(name);
    for (ConfigEventListener listener : listeners) {
        listener.onConfigLoaded(cfg);
    }
    return cfg;
}

public void registerConfigEventListener(ConfigEventListener listener) {
    listeners.add(listener);
}

public void removeConfigEventListener(ConfigEventListener listener) {
    listeners.remove(listener);
}

public void update(String name, int newVersion,
    Map<String, String> properties) {
    for (ConfigEventListener listener : listeners) {
        listener.onConfigUpdated(name, newVersion, properties);
    }
}
}

```

提示

规避死锁的常见方法：

- 粗锁法（Coarsen-grained Lock）——使用一个粗粒度的锁代替多个锁。
- 锁排序法（Lock Ordering）——相关线程使用全局统一的顺序申请锁。
- 使用 `ReentrantLock.tryLock(long, TimeUnit)` 来申请锁。
- 使用开放调用（Open Call）——在调用外部方法时不加锁。
- 使用锁的替代品。

规避死锁的另外一种“终极”方法就是不使用锁！第 6 章以及前面章节我们介绍了一些锁的替代品（无状态对象、线程特有对象以及 `volatile` 关键字等）。在条件允许的情况下使用这些替代品在保障线程安全的前提下不仅能够避免锁的开销，还能够直接避免死锁！

7.1.3 死锁的恢复

前面我们介绍的是如何在代码这一层规避死锁的产生，即防患于未然，那么万一死锁已然产生，如何将其解除呢？这就是死锁的故障恢复问题。如果代码中使用的是内部锁或者使用的是显式锁而锁的申请是通过 `Lock.lock()` 调用实现的，那么这些锁的使用所导致的死锁故障是不可恢复的，而我们唯一能够做的就是重启 Java 虚拟机。如果代码中使用的

是显式锁且锁的申请是通过 `Lock.lockInterruptibly()` 调用实现的,那么这些锁的使用所导致的死锁理论上是可恢复的,但是,死锁的恢复实际可操作性并不强——进行恢复的尝试可能是徒劳的(故障线程可无法响应中断)且有害的(可能导致其他线程活性故障)!尽管如此,我们仍然探讨这个问题,是因为它有助于我们进一步理解线程的中断机制。

注意

由于导致死锁的线程的不可控性(比如第三方软件启动的线程),因此死锁恢复的实际可操作性并不强:对死锁进行的故障恢复尝试可能是徒劳的(故障线程可无法响应中断)且有害的(可能导致活锁等问题)。

死锁的自动恢复有赖于线程的中断机制,其基本思想是:定义一个工作者线程 `DeadlockDetector` 专门用于死锁检测与恢复,如清单 7-14 所示。该线程定期检测系统中是否存在死锁,若检测到死锁,则随机选取一个死锁线程并给其发送中断。该中断使得一个任意的死锁线程(目标线程)被 Java 虚拟机唤醒,从而使其抛出 `InterruptedException` 异常。这使得目标线程不再等待它本来永远也无法申请到的资源,从而破坏了死锁产生的必要条件中的“占用并等待资源”中的“等待资源”部分。目标线程则通过对 `InterruptedException` 进行处理的方式来响应中断:目标线程捕获 `InterruptedException` 异常后将其已经持有的资源(锁)主动释放掉,这相当于破坏了死锁产生的必要条件中的“占用并等待资源”中的“占用资源”部分。接着, `DeadlockDetector` 继续检测系统中是否仍然存在死锁,若存在,则继续选中一个任意的死锁线程并给其发送中断,直到系统中不再存在死锁。

清理 7-14 死锁检测与恢复线程源码

```
public class DeadlockDetector extends Thread {
    static final ThreadMXBean tmb = ManagementFactory.getThreadMXBean();
    /**
     * 检测周期(单位为毫秒)
     */
    private final long monitorInterval;

    public DeadlockDetector(long monitorInterval) {
        super("DeadLockDetector");
        setDaemon(true);
        this.monitorInterval = monitorInterval;
    }

    public DeadlockDetector() {
        this(2000);
    }

    public static ThreadInfo[] findDeadlockedThreads() {
        long[] ids = tmb.findDeadlockedThreads();
        return null == tmb.findDeadlockedThreads() ?
```



```

        new ThreadInfo[0] : tmb.getThreadInfo(ids);
    }

    public static Thread findThreadById(long threadId) {
        for (Thread thread : Thread.getAllStackTraces().keySet()) {
            if (thread.getId() == threadId) {
                return thread;
            }
        }
        return null;
    }

    public static boolean interruptThread(long threadID) {
        Thread thread = findThreadById(threadID);
        if (null != thread) {
            thread.interrupt();
            return true;
        }
        return false;
    }

    @Override
    public void run() {
        ThreadInfo[] threadInfoList;
        ThreadInfo ti;
        int i = 0;
        try {
            for (;;) {
                // 检测系统中是否存在死锁
                threadInfoList = DeadlockDetector.findDeadlockedThreads();
                if (threadInfoList.length > 0) {
                    // 选取一个任意的死锁线程
                    ti = threadInfoList[i++ % threadInfoList.length];
                    Debug.error("Deadlock detected, trying to recover"
                        + " by interrupting\n thread(%d,%s)\n",
                        ti.getThreadId(),
                        ti.getThreadName());
                    // 给选中的死锁线程发送中断
                    DeadlockDetector.interruptThread(ti.getThreadId());
                    continue;
                } else {
                    Debug.info("No deadlock found!");
                    i = 0;
                }
                // for 循环结束
                Thread.sleep(monitorInterval);
            }
        } catch (InterruptedException e) {
    }

```

```

    // 什么也不做
    ;
}
}
}

```

DeadlockDetector 是通过 `java.lang.management.ThreadMXBean.findDeadlockedThreads()` 调用来实现死锁检测的。`ThreadMXBean.findDeadlockedThreads()` 能够返回一组死锁线程的线程编号。`ThreadMXBean` 类是 JMX (Java Management Extension) API 的一部分, 因此其提供的功能也可以通过 `jconsole`、`jvisualvm` 手工调用⁷。

清单 7-5 中的哲学家模型由于是通过 `ReentrantLock.lock()` 申请显式锁的, 因此它无法响应中断, 也就无法支持死锁的自动恢复。因此为了展示死锁恢复的效果, 我们需要将其改造为如清单 7-15 所示的代码。

清单 7-15 支持死锁恢复的 `AbstractPhilosopher` 子类 `RecoverablePhilosopher`

```

public class RecoverablePhilosopher extends BuggyLckBasedPhilosopher {

    public RecoverablePhilosopher(int id, Chopstick left, Chopstick right) {
        super(id, left, right);
    }

    @Override
    protected boolean pickUpChopstick(Chopstick chopstick) {
        final ReentrantLock lock = LOCK_MAP.get(chopstick);
        try {
            lock.lockInterruptibly();
        } catch (InterruptedException e) {
            // 使当前线程释放其已持有的锁
            Debug.info("%s detected interrupt.", Thread.currentThread().getName());
            Chopstick theOtherChopstick = chopstick == left ? right : left;
            theOtherChopstick.putDown();
            LOCK_MAP.get(theOtherChopstick).unlock();
            return false;
        }
        try {
            Debug.info(
                "%s is picking up %s on his %s...%n",
                this, chopstick, chopstick == left ? "left" : "right");

            chopstick.pickUp();

```

7 `jvisualvm` 中需要先安装 VisualVM-MBeans 插件才支持手工调用 JMX API, 参见: https://visualvm.java.net/mbeans_tab.html。

```

    } catch (Exception e) {
        // 不大可能走到这里
        e.printStackTrace();
        lock.unlock();
        return false;
    }
    return true;
}
}

```

这里, `pickUpChopstick` 方法在捕获到 `lock.lockInterruptibly()` 抛出的 `InterruptedException` 后, 主动将当前线程已持有的锁释放掉 (即放下当前哲学家已持有的筷子)。利用这个改造后的哲学家模型, 我们就可以再现死锁的自动恢复的效果, 如清单 7-16 所示。

清单 7-16 死锁自动恢复 Demo

```

public class DeadlockRecoveryDemo {

    public static void main(String[] args) throws Exception {
        // 创建并启动死锁检测与恢复线程
        new DeadlockDetector().start();
        // 指定 RecoverablePhilosopher 为哲学家模型实现类
        System.setProperty("x.philo.impl",
            "RecoverablePhilosopher");
        // 启动哲学家就餐问题模拟程序
        DiningPhilosopherProblem.main(args);
    }
}

```

运行上述程序, 我们可以发现当死锁产生的时候, `DeadlockDetector` 能够自动侦测到并试图进行自动恢复。但是, 恢复之后故障又很快重新出现了, 接着又是自动恢复……由此可见, 死锁自动恢复的实际意义并不大。

首先, 死锁的自动恢复有赖于死锁的线程能够响应中断。以 `RecoverablePhilosopher` (清单 7-15) 为例, 如果我们在代码开发与维护过程中能够意识到它是可能导致死锁的, 那么我们应该采取的措施是规避死锁 (防患未然) 而不是使其支持死锁的自动恢复 (为亡羊补牢做准备); 相反, 如果我们未能事先意识到死锁这个问题, 那么这个类的相关方法可能根本无法响应中断, 或者能够响应中断但是其响应的结果却未必是 `DeadlockDetector` 所期望的——释放其已持有的资源。

其次, 自动恢复尝试可能导致新的问题。例如, 如果 `RecoverablePhilosopher` (清单 7-15) 对中断的响应方式是仅仅保留中断标记而并不释放其已持有的资源, 即 `RecoverablePhilosopher.pickUpChopstick` 方法对 `InterruptedException` 异常的处理逻辑仅仅

是调用 `Thread.currentThread().interrupt()` 以保留中断标记,那么尝试对这样的死锁线程进行恢复非但不能达到预期效果,反而会造成相应线程一直在尝试申请锁而一直无法申请成功,即产生活锁(7.4节会介绍这个概念)!

7.2 沉睡不醒的睡美人: 锁死

等待线程由于唤醒其所需的条件永远无法成立,或者其他线程无法唤醒这个线程而一直处于非运行状态(线程并未终止)导致其任务一直无法进展,那么我们就称这个线程被锁死(Lockout)。锁死就好比睡美人的故事中睡美人醒来的前提是她要得到王子的亲吻。但是如果王子无法亲吻她(比如王子“挂了”……),那么睡美人将一直沉睡!

有些资料可能将锁死与死锁混为一谈,这样做表面看来似乎没有什么害处,毕竟锁死与死锁有着共同的外在表现——故障线程一直处于非运行状态而使得其任务无法进展。但是,锁死与死锁的产生条件是不同的,即便是在产生死锁的所有必要条件都不成立的情况下(此时死锁不可能产生),锁死仍然可能出现。因此,“对付”死锁的办法未必能够用来“对付”锁死,将锁死与死锁区分开来是有必要的。按照锁死产生的条件来分,锁死包括信号丢失锁死和嵌套监视器锁死。

7.2.1 信号丢失锁死

信号丢失锁死是由于没有相应的通知线程来唤醒等待线程而使等待线程一直处于等待状态的一种活性故障。

信号丢失锁死的一个典型例子是等待线程在执行 `Object.wait()/Condition.await()` 前没有对保护条件进行判断,而此时保护条件实际上可能已然成立,然而此后可能并无其他线程更新相应保护条件涉及的共享变量使其成立并通知等待线程,这就使得等待线程一直处于等待状态,从而使其任务一直无法进展。这就是我们在第5章中强调 `Object.wait()/Condition.await()` 必须放在一个循环语句中的原因之一。信号丢失锁死的另外一个常见例子是 `CountDownLatch.countDown()` 调用没有放在 `finally` 块中导致 `CountDownLatch.await()` 的执行线程一直处于等待状态,从而使其任务一直无法进展。

7.2.2 嵌套监视器锁死

嵌套监视器锁死(Nested Monitor Lockout)是嵌套锁导致等待线程永远无法被唤醒的一种活性故障。假设某个程序使用如图7-4所示的受保护方法及相应的通知方法来实现“等

待/通知”。我们知道，等待线程在其执行到 `monitorY.wait()` 的时候会被暂停并且其所持有的锁 `monitorY` 会被释放，但是等待线程所持有的外层锁 `monitorX` 并不会因此（`Object.wait()` 调用）而被释放。通知线程在调用 `monitorY.notifyAll()` 来唤醒等待线程时需要持有相应的锁 `monitorY`，但是由于 `monitorY` 所引导的临界区位于 `monitorX` 引导的临界区之内，因此通知线程必须先持有外层锁 `monitorX`。而通知线程执行通知方法的时候，其所需申请的 `monitorX` 可能正好被等待线程所持有，因此通知线程无法唤醒等待线程。而等待线程只有在被唤醒之后（退出内层临界区）才能够释放其持有的外层锁 `monitorX`。于是，通知线程始终无法获得锁 `monitorX`，从而无法通过 `monitorY.notifyAll()` 调用来唤醒等待线程，这使得等待线程一直处于非运行状态（这里是 `BLOCKED` 状态）。这种由于嵌套锁导致通知线程始终无法唤醒等待线程的活性故障就被称为嵌套监视器锁死。

受保护方法	通知方法
<pre>synchronized (monitorX) { // ... synchronized (monitorY) { while (!somethingOK) { monitorY.wait(); } // 此处执行目标动作 ... } // ... }</pre>	<pre>synchronized (monitorX) { // ... synchronized (monitorY) { // ... somethingOK = true; monitorY.notifyAll(); } // ... }</pre>

图 7-4 嵌套监视器锁死的代码特征

我们实际接触到的代码可能并不像图 7-4 所示的那样特征明显。清单 7-17 展示了一个嵌套监视器锁死 Demo，这是一个简单的生产者—消费者实例，其中 `main` 线程是生产者线程，而工作者线程（`WorkerThread` 实例）是消费者线程。从表面上看，这个 Demo 并没有符合图 7-4 所示的特征，但是运行这个 Demo 可以发现该程序很快就“冻住”了（可能没有任何输出）。而查看该程序的线程转储，我们并未发现有死锁。

清单 7-17 嵌套监视器锁死 Demo

```
public class NestedMonitorLockoutDemo {  
    private final BlockingQueue<String> queue = new ArrayBlockingQueue<String>(10);  
    private int processed = 0;  
    private int accepted = 0;  
  
    public static void main(String[] args) throws InterruptedException {  
        NestedMonitorLockoutDemo u = new NestedMonitorLockoutDemo();  
        u.start();  
        int i = 0;  
        while (i-- < 100000) {
```

```

        u.accept("message" + i);
        Tools.randomPause(100);
    }

}

public synchronized void accept(String message) throws InterruptedException {
    // 不要在临界区内调用 BlockingQueue 的阻塞方法! 那样会导致嵌套监视器锁死
    queue.put(message);
    accepted++;
}

protected synchronized void doProcess() throws InterruptedException {
    // 不要在临界区内调用 BlockingQueue 的阻塞方法! 那样会导致嵌套监视器锁死
    String msg = queue.take();
    System.out.println("Process:" + msg);
    processed++;
}

public void start() {
    new WorkerThread().start();
}

public synchronized int[] getStat() {
    return new int[] { accepted, processed };
}

class WorkerThread extends Thread {
    @Override
    public void run() {
        try {
            while (true) {
                doProcess();
            }
        } catch (InterruptedException e) {
            ;
        }
    }
}

```

尽管如此, 查看该程序的线程转储我们不难发现, 消费者线程 Thread-0(WorkerThread 的实例) 持有一个标识号为 0x00000000d72f9b80 的内部锁 (NestedMonitorLockoutDemo 实例) 并等待 (线程生命周期状态为 WAITING):

```

"Thread-0" #9 prio=5 os_prio=0 tid=0x00007fd0fc13f800 nid=0x29ab waiting on
condition [0x00007fd0dae26000]

```

```

java.lang.Thread.State: WAITING (parking)
    at sun.misc.Unsafe.park(Native Method)
    - parking to wait for <0x00000000d72fb1c0> (a java.lang.AbstractQueuedSynchronizer$ConditionObject)
    at java.lang.LockSupport.park(LockSupport.java:175)
    at java.lang.AbstractQueuedSynchronizer$ConditionObject.await(AbstractQueuedSynchronizer.java:2039)
    at java.util.concurrent.ArrayBlockingQueue.take(ArrayBlockingQueue.java:403)
    at mtia.NestedMonitorLockoutDemo.doProcess(NestedMonitorLockoutDemo.java:32)
    - locked <0x00000000d72f9b80> (a mtia.NestedMonitorLockoutDemo)
    at mtia.NestedMonitorLockoutDemo$WorkerThread.run(NestedMonitorLockoutDemo.java:50)

Locked ownable synchronizers:
    - None

```

结合代码来看，不难得知这个等待实际上是等待阻塞队列 queue 非空（即生产者生产了新的产品），如图 7-5 所示（图中的 notEmpty 是一个 Condition 实例）。

```

public E take() throws InterruptedException {
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly();
    try {
        try {
            while (count == 0)
                notEmpty.await();
        } catch (InterruptedException ie) {
            notEmpty.signal(); // propagate to non interrupted thread
            throw ie;
        }
        E x = extract();
        return x;
    } finally {
        lock.unlock();
    }
}

```

图 7-5 ArrayBlockingQueue.take()源码

相应地，ArrayBlockingQueue.put()会在队列非空的情况下通知相应的等待线程以唤醒相应的消费者线程，如图 7-6 所示（图中的 notFull 是一个 Condition 实例）。

```

public void put(E e) throws InterruptedException {
    if (e == null) throw new NullPointerException();
    final E[] items = this.items;
    final ReentrantLock lock = this.lock;
    lock.lockInterruptibly();
    try {
        try {
            while (count == items.length)
                notFull.await();
        } catch (InterruptedException ie) {
            notFull.signal(); // propagate to non interrupted thread
            throw ie;
        }
        insert(e);
    } finally {
        lock.unlock();
    }
}

```

图 7-6 ArrayBlockingQueue.put()源码

再看相应的生产者线程（main 线程）的情况，NestedMonitorLockoutDemo.accept 方法是一个同步方法，因此生产者在调用该方法将其生产的“产品”存入队列时需要先申请该方法所需的内部锁（NestedMonitorLockoutDemo 实例），如下线程转储片段所示：

```

"main" #1 prio=5 os_prio=0 tid=0x00007fd0fc00b000 nid=0x2996 waiting for monitor
entry [0x00007fd10447d000]
  java.lang.Thread.State: BLOCKED (on object monitor)
    at mtia.NestedMonitorLockoutDemo.accept(NestedMonitorLockoutDemo.java:26)
    - waiting to lock <0x00000000d72f9b80> (a mtia.NestedMonitorLockoutDemo)
    at mtia.NestedMonitorLockoutDemo.main(NestedMonitorLockoutDemo.java:18)

Locked ownable synchronizers:
  - None

```

可见，生产者线程正在等待标识号为 0x00000000d72f9b80 的内部锁（NestedMonitorLockoutDemo 实例），而这个锁恰好被消费者线程 Thread-0 所持有，因此生产者无法执行 queue.put(message)，也就无法将“产品”存入队列（相当于无法生产“产品”），而 queue.put(message)无法被执行就意味着消费者线程无法被唤醒（因为队列一直是空的）！

从等待/通知的角度来看，这个生产者线程相当于通知线程，它无法生产“产品”就

不会通知等待线程（消费者线程）队列非空，那么等待线程（Thread-0）就会一直处于等待状态，而等待线程一直处于等待状态则会导致其持有的内部锁（0x00000000d72f9b80，NestedMonitorLockoutDemo 当前实例对应的内部锁）一直无法被释放。这样，生产者线程便永远无法生产“产品”，而消费者线程也永远处于等待状态。

在这个 Demo 中，ArrayBlockingQueue.take()/ArrayBlockingQueue.put(E)内部使用的锁连同 NestedMonitorLockoutDemo.doProcess 方法/accept 方法自身使用的内部锁事实上形成了图 7-4 所示的代码特征——在嵌套锁的内层临界区中调用 Object.wait()/notify()/notifyAll() 或者 Condition.await()/signal()/signalAll()。因此，我们看到的“冻住”现象实际上是嵌套监视器锁死，而不是死锁！

从死锁产生的必要条件角度出发，我们不难看出嵌套监视器锁死与死锁的区别。尽管上述 Demo 也存在嵌套锁，但是由于其中的两个线程（main 线程和 Thread-0）都是按照全局统一的顺序（先申请 NestedMonitorLockoutDemo 当前实例对应的内部锁，再申请 ArrayBlockingQueue 实例内部的显式锁）来申请锁的，这相当于采取前文提到的“锁排序法”来规避死锁，因此该 Demo 不可能出现死锁。尽管如此，该 Demo 仍然出现锁死。在本质上，嵌套监视器锁死是由于通知线程无法获得锁，导致其无法唤醒等待线程，最终使等待线程永远处于等待状态的活性故障；而死锁是由于所有故障线程都无法获得其所需的锁而导致的活性故障。

在上述 Demo 中，我们只需要将 ArrayBlockingQueue.take()调用挪到 doProcess 方法之外就可以规避嵌套监视器锁死了，如下代码片段所示：

```
protected synchronized void doProcess(String msg) throws InterruptedException {
    System.out.println("Process:" + msg);
    processed++;
}

class WorkerThread extends Thread {
    @Override
    public void run() {
        try {
            String msg;
            while (true) {
                msg = queue.take();
                doProcess(msg);
            }
        } catch (InterruptedException e) {
            ;
        }
    }
}
```

7.3 巧妇难为无米之炊：线程饥饿

线程饥饿 (Thread Starvation) 是指线程一直无法获得其所需的资源而导致其任务一直无法进展的一种活性故障。线程饥饿相当于俗话说的“巧妇 (线程) 难为无米 (资源) 之炊 (任务)”。

线程饥饿的一个典型例子是在高争用的环境下使用非公平模式 (Non-fair mode) 的读写锁 (ReentrantReadWriteLock)。例如, 在 Web 应用程序中使用 ReentrantReadWriteLock 来保护配置数据, 业务线程可能不断地申请读写锁的读锁来读取配置数据, 由于默认情况下 ReentrantReadWriteLock 的锁调度采用非公平调度模式, 因此如果这些业务线程对锁的争用程度比较高, 那么系统管理模块试图更新配置数据的时候就可能遇到这样的情形: 相应的读锁总是被业务线程抢先占有 (非公平锁调度可能导致的结果) 导致系统管理模块的线程始终无法获得相应的写锁, 从而使其一直无法更新配置数据。因此, 尽管非公平锁可以支持更高的吞吐率, 但是它也可能导致某些线程总是无法获取其所需的资源 (锁), 即导致线程饥饿。

把锁看作一种资源, 那么我们不难发现死锁也是一种线程饥饿。死锁的结果是故障线程都无法获得其所需的全部锁中的一个锁, 从而使其任务一直无法进展, 这相当于线程无法获得其所需的全部资源 (锁) 而使得其任务一直无法进展, 即产生了线程饥饿。由于线程饥饿的产生条件是一个 (或者多个) 线程始终无法获取其所需的资源, 显然这个条件的满足并不意味着死锁产生的必要条件 (这还仅是必要条件, 而不是充分条件) 的满足, 因此线程饥饿并不会导致死锁。

线程饥饿涉及的线程, 其生命周期状态不一定是 WAITING 或者 BLOCKED 状态, 其状态也可能是 RUNNING (这说明涉及的线程一直在申请其所需的资源), 这时饥饿就演变成 7.4 节介绍的活锁。

7.4 屡战屡败, 屡败屡战: 活锁

活锁 (Livelock) 是指线程一直处于运行状态, 但是其任务却一直无法进展的一种活性故障。也就是说, 产生活锁的线程一直在做无用功, 这就好比小猫追着自己的尾巴咬, 虽然它一直在咬, 但是却一直咬不到自己的尾巴!

线程在争取其所需的资源过程中如果“屡战屡败，屡败屡战”——线程一直在申请其所需的资源而一直未申请成功，那么此时线程饥饿实际上就演变成活锁。

7.5 本章小结

本章介绍了常见的线程活性故障以及相应的规避措施。本章知识结构如图 7-7 所示。

死锁会导致相关线程一直被暂停使得其任务无法进展。产生死锁的必要条件包括：资源互斥、资源不可抢夺、占用并等待资源以及循环等待资源。我们可以通过查看线程转储手工检测死锁，也可以利用 `ThreadMXBean.findDeadlockedThreads()` 方法进行死锁的自动检测。死锁的规避方法包括：粗锁法（使用一个粗粒度的锁代替多个锁）、锁排序法（相关线程使用全局统一的顺序申请锁）、使用 `ReentrantLock.tryLock(long, TimeUnit)` 来申请锁、使用开放调用（在调用外部方法时不加锁）以及使用锁的替代品。使用内部锁或者使用 `lock.lock()` 申请的显式锁导致的死锁是无法恢复的；使用 `lock.lockInterruptibly()` 申请的显式锁导致的死锁理论上可恢复的，但实际可操作性不强——自动恢复的尝试可能是徒劳且有害的（导致活锁）。

锁死是等待线程由于某种原因一直无法被唤醒而导致其任务无法进展的一种活性故障。信号丢失锁死是由于没有相应的通知线程来唤醒等待线程而使等待线程一直处于等待状态的一种活性故障。嵌套监视器锁死是嵌套锁导致通知线程无法获得其为唤醒等待线程所需的锁从而使其无法唤醒等待线程，最终使得通知线程与等待线程都一直处于等待状态的一种活性故障。嵌套监视器锁死可以通过查看线程转储进行检测。为规避嵌套监视器锁死，我们应该避免在嵌套锁的内层临界区内实现等待/通知。

线程饥饿指线程一直无法获得其所需的资源而导致其任务一直无法进展的一种活性故障。把锁看成一种资源，那么死锁可被看作一种线程饥饿。饥饿可能演变成活锁。

活锁是线程一直在做无用功而使其任务一直无法进展的一种活性故障。试图进行死锁故障恢复可能导致活锁。

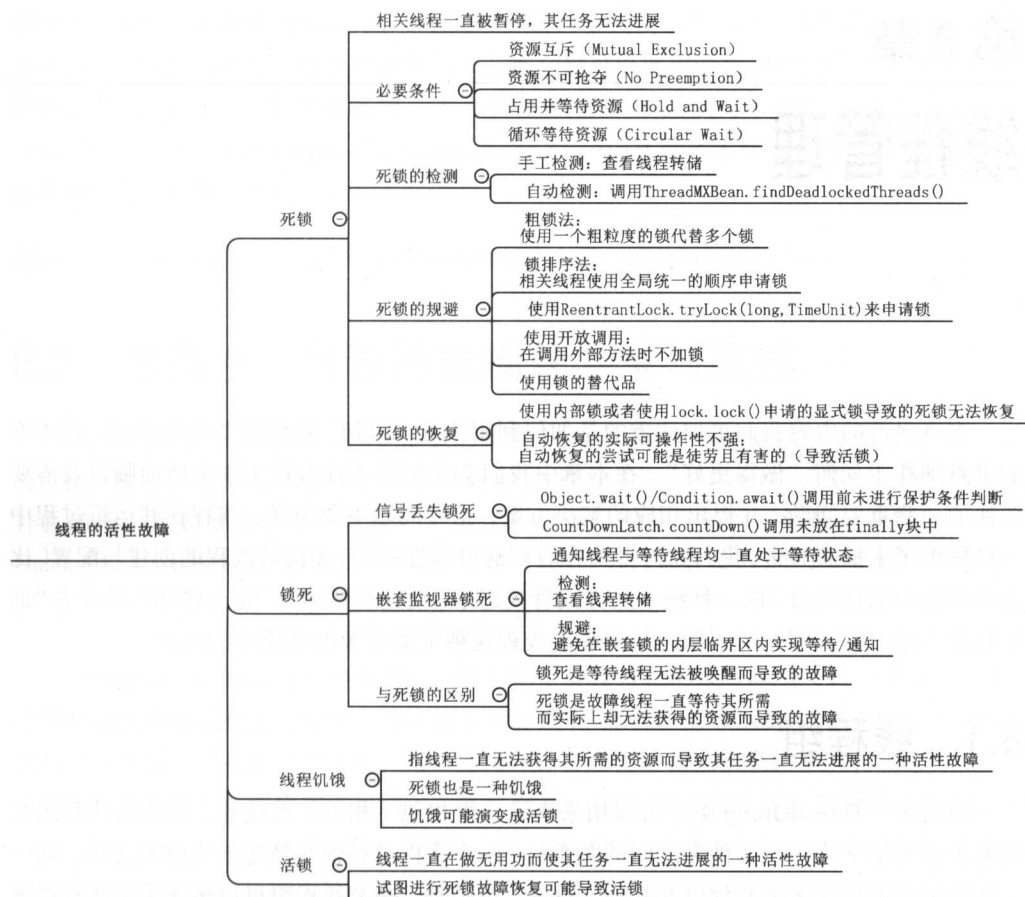


图 7-7 本章知识结构图

第 8 章

线程管理

本章之前的内容我们更加注重的是如何利用线程“做到”我们想要做的事情，而本章的重点则在于如何“做得更好”。在本章中我们会介绍多线程编程实战中所面临以及需要关注的一些重要问题，并提出相应的解决方案。这些问题主要包括：线程在其运行过程中一旦抛出了未捕获异常，我们如何得知并应对的可靠性问题；如何将线程的创建与配置（比如设置线程的优先级）以一种统一的方式管控起来，使这些线程从“散兵游勇”提升为“训练有素”的“正规军”的问题；如何提高线程这种宝贵资源的利用率的问题。

8.1 线程组

线程组（`ThreadGroup` 类）可以用来表示一组相似（相关）的线程。线程与线程组之间的关系类似于文件与文件夹之间的关系——一个文件总是位于特定的文件夹之中，而一个文件夹可以包含多个文件以及其他文件夹。类似地，一个线程组可以包含多个线程以及其他线程组。一个线程组包含其他线程组的时候，该线程组被称为这些线程组的父线程组。`Thread` 类有几个构造器允许我们在创建线程的时候指定线程所属的线程组。如果创建线程的时候我们没有指定线程组，那么这个线程就属于其父线程（即当前线程）所属的线程组。由于 Java 虚拟机在创建 `main` 线程（Java 平台中所有线程的父线程）时会为其指定一个线程组，因此 Java 平台中的任何一个线程都有一个线程组与之关联，这个线程组可以通过 `Thread.getThreadGroup()` 调用来获取。

`ThreadGroup` 最初是出于安全的考虑被设计用来隔离（区分）不同的 Applet 的。然而，`ThreadGroup` 并未实现这一预期目标，并且它所实现的许多方法是有缺陷的，另外这些方法也不是很常用¹。一些遗留（Legacy）系统中可能还存在对 `ThreadGroup` 的使用。在新开发的系统中，如果我们需要将一些线程归结为一组，那么可以考虑简单的办法：将这些线

1 详见：《Effective Java》第 1 版的“Item 53: Avoid thread groups”。

程存入一个数组或者集合对象中,当然这样处理可能需要注意内存泄漏问题。如果仅仅是为了将一些线程与另外一些线程区分开来,那么也可以使用线程名称的命名规则来实现。例如,第4章第1个实战案例(大文件下载器)中我们将下载工作者线程命名为“downloader-0”、“downloader-1”、“downloader-2”……从这些线程名我们就可以看出相应线程的功能是相同(相近)的。

提示

多数情况下,我们可以忽略线程组这一概念以及线程组的存在。

8.2 可靠性: 线程的未捕获异常与监控

如果线程的 run 方法抛出未被捕获的异常(Uncaught Exception),那么随着 run 方法的退出,相应的线程也提前终止。对于线程的这种异常终止,我们如何得知并做出可能的补救动作,例如重新创建并启动一个替代线程呢?JDK 1.5 为了解决这个问题引入了 UncaughtExceptionHandler 接口。该接口是在 Thread 类内部定义的,它只定义了一个方法:

```
void uncaughtException(Thread t, Throwable e)
```

uncaughtException 方法中的两个参数包括了异常终止的线程本身(对应第1个参数)以及导致线程提前终止的异常(对应第2个参数)。那么,在 uncaughtException 方法当中我们就可以做一些有意义的事情,比如将线程异常终止的相关信息记录到日志文件中,甚至于为异常终止的线程创建并启动一个替代线程。设 thread 为任意一个线程,eh 为任意一个 UncaughtExceptionHandler 实例,那么我们可以在启动 thread 前通过调用 thread.setUncaughtExceptionHandler(eh)来为 thread 关联一个 UncaughtExceptionHandler。当 thread 抛出未被捕获的异常后 thread.run()返回,接着 thread 会在其终止前调用 eh.uncaughtException 方法。

清单 8-1 展示了一个利用 UncaughtExceptionHandler 实现线程监控的例子。在这个例子中,系统的某个重要服务(ThreadMonitorDemo)内部维护了一个工作者线程(WorkerThread)用于实现该服务的核心功能。因此,一旦这个工作者线程由于某些未捕获的异常(比如 NullPointerException)而提前终止,那么我们需要在第一时间得到“通知”,并为该线程创建并启动一个替代线程来接替其完成其任务,以保障该服务的可靠性。这个接替的过程就是通过 UncaughtExceptionHandler 实现的:ThreadMonitor.uncaughtException 方法会重新将工作者线程的启动标记 init 置为 false,并再次调用 init 方法来创建并启动一个新的工作者线程,用于接替异常中止的工作者线程。

清单 8-1 使用 UncaughtExceptionHandler 实现线程监控

```

public class ThreadMonitorDemo {
    volatile boolean initied = false;
    static int threadIndex = 0;
    final static Logger LOGGER = Logger.getAnonymousLogger();
    final BlockingQueue<String> channel = new ArrayBlockingQueue<String>(100);

    public static void main(String[] args) throws InterruptedException {
        ThreadMonitorDemo demo = new ThreadMonitorDemo();
        demo.init();
        for (int i = 0; i < 100; i++) {
            demo.service("test-" + i);
        }
        Thread.sleep(2000);
        System.exit(0);
    }

    public synchronized void init() {
        if (initied) {
            return;
        }
        Debug.info("init...");
        WorkerThread t = new WorkerThread();
        t.setName("Worker0-" + threadIndex++);
        // 为线程 t 关联一个 UncaughtExceptionHandler
        t.setUncaughtExceptionHandler(new ThreadMonitor());
        t.start();
        initied = true;
    }

    public void service(String message) throws InterruptedException {
        channel.put(message);
    }

    private class ThreadMonitor implements Thread.UncaughtExceptionHandler {
        @Override
        public void uncaughtException(Thread t, Throwable e) {
            Debug.info("Current thread is `t`: %s, it is still alive: %s",
                Thread.currentThread() == t, t.isAlive());

            // 将线程异常终止的相关信息记录到日志中
            String threadInfo = t.getName();
            LOGGER.log(Level.SEVERE, threadInfo + " terminated:", e);

            // 创建并启动替代线程
            LOGGER.info("About to restart " + threadInfo);
            // 重置线程启动标记

```

```

    initied = false;
    init();
}

} // 类 ThreadMonitor 定义结束

private class WokrerThread extends Thread {
    @Override
    public void run() {
        Debug.info("Do something important...");
        String msg;
        try {
            for (;;) {
                msg = channel.take();
                process(msg);
            }
        } catch (InterruptedException e) {
            // 什么也不做
        }
    }

    private void process(String message) {
        Debug.info(message);
        // 模拟随机性异常
        if ((int) (Math.random() * 100) < 2) {
            throw new RuntimeException("test");
        }
        Tools.randomPause(100);
    }
} // 类 ThreadMonitorDemo 定义结束
}

```

运行上述程序，我们可以看到类似如下的输出（省略部分输出）：

```

[2016-11-29 19:03:04.556] [INFO] [main]:init...
[2016-11-29 19:03:04.560] [INFO] [Worker0-0]:Do something important...
[2016-11-29 19:03:04.561] [INFO] [Worker0-0]:test-0
[2016-11-29 19:03:04.568] [INFO] [Worker0-0]:test-1
【此处省略部分输出】
[2016-11-29 19:03:04.816] [INFO] [Worker0-0]:current thread is `t`:true, it is
still alive:true
Nov 29, 2016 7:03:04 PM mtia.ThreadMonitorDemo$ThreadMonitor uncaughtException
SEVERE: Worker0-0 terminated:
java.lang.RuntimeException: test
    at mtia.ThreadMonitorDemo$WokrerThread.process(ThreadMonitorDemo.java:84)
    at mtia.ThreadMonitorDemo$WokrerThread.run(ThreadMonitorDemo.java:72)

Nov 29, 2016 7:03:04 PM mtia.ThreadMonitorDemo$ThreadMonitor uncaughtException

```


INFO: About to restart Worker0-0

[2016-11-29 19:03:04.834] [INFO] [Worker0-0]:init...

[2016-11-29 19:03:04.834] [INFO] [Worker0-1]:Do something important...

[2016-11-29 19:03:04.835] [INFO] [Worker0-1]:test-8

[2016-11-29 19:03:04.904] [INFO] [Worker0-1]:test-9

【此处省略部分输出】

[2016-11-29 19:03:09.299] [INFO] [Worker0-1]:test-99

可见，工作者线程（WorkerThread）中途的确异常终止过，但是由于我们在侦测到该线程异常终止的时候创建了相应的替代线程，因此该线程的异常终止并没有影响 ThreadMonitorDemo 继续对外提供服务，从而使 ThreadMonitorDemo 的可靠性得以保障。另外，从上述输出中可以看出，UncaughtExceptionHandler.uncaughtException 方法是执行在抛出异常 e 的线程 t 之中的，在执行 UncaughtExceptionHandler.uncaughtException 方法的时候线程 t 还是存活的（Live），UncaughtExceptionHandler.uncaughtException 方法返回之后线程 t 就终止了。

线程组本身也实现了 UncaughtExceptionHandler 接口。如果一个线程没有关联的 UncaughtExceptionHandler 实例，那么该线程异常终止前其所属线程组的 uncaughtException 方法会被调用。线程组的 uncaughtException 方法会调用其父线程组的 uncaughtException 方法并传递同样的两个参数（t 和 e）。如果一个线程组没有其父线程组²，那么线程组的 uncaughtException 方法会调用默认 UncaughtExceptionHandler 的 uncaughtException 方法来处理线程的异常终止。默认 UncaughtExceptionHandler 适用于所有线程，即任何一个线程异常终止时默认 UncaughtExceptionHandler 都有可能被调用。Thread.setDefaultUncaughtExceptionHandler 方法可用来指定默认 UncaughtExceptionHandler。针对一个线程的异常终止，该线程所关联的 UncaughtExceptionHandler 实例、该线程所在的线程组以及默认 UncaughtExceptionHandler 之中只有一个 UncaughtExceptionHandler 实例会被选中。UncaughtExceptionHandler 实例的选择优先级如图 8-1 所示。

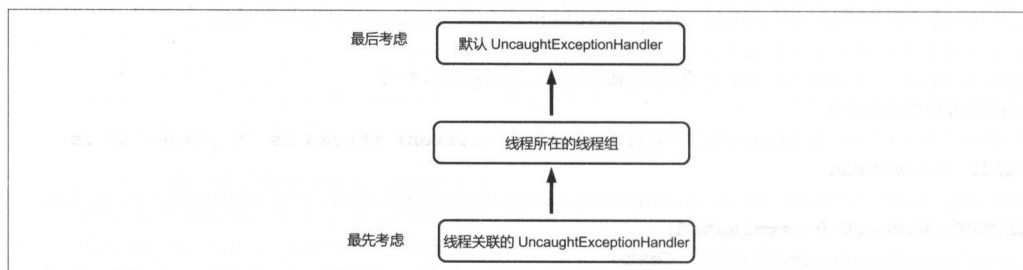


图 8-1 UncaughtExceptionHandler 实例的选择优先级示意图

2 只有最顶层的线程组没有其父线程组，因此一个 Java 虚拟机中只有一个线程组没有其父线程组。

清单 8-2 展示了默认 `UncaughtExceptionHandler` 在 Web 应用中的使用。在该例子中，我们先在 `ServletContextListener.contextInitialized` 方法中设置了默认 `UncaughtExceptionHandler`，接着再启动该 Web 应用所需的若干工作者线程。该默认 `UncaughtExceptionHandler` 对线程异常终止的处理仅仅是将抛出异常的线程的相关信息记录到日志文件中。当然，如果有特别的需要，我们也可以在該 `UncaughtExceptionHandler` 中向告警子系统发送相关告警信息，甚至发送相关的短信。

清单 8-2 在 Web 应用中使用默认 `UncaughtExceptionHandler`

```
public class AppListener implements ServletContextListener {
    final static Logger LOGGER = Logger.getAnonymousLogger();

    @Override
    public void contextInitialized(ServletContextEvent contextEvent) {
        // 设置默认 UncaughtExceptionHandler
        UncaughtExceptionHandler ueh = new LoggingUncaughtExceptionHandler();
        Thread.setDefaultUncaughtExceptionHandler(ueh);

        // 启动若干工作者线程
        startServices();
    }

    static class LoggingUncaughtExceptionHandler implements
        UncaughtExceptionHandler {
        @Override
        public void uncaughtException(Thread t, Throwable e) {
            String threadInfo = "Thread[" + t.getName() + ", " + t.getId() + ", "
                + t.getThreadGroup().getName() + ", @" + t.hashCode() + "];"

            // 将线程异常终止的相关信息记录到日志中
            LOGGER.log(Level.SEVERE, threadInfo + " terminated:", e);
        }
    }

    protected void startServices() {
        // 省略其他代码
    }

    protected void stopServices() {
        // 省略其他代码
    }

    @Override
    public void contextDestroyed(ServletContextEvent contextEvent) {
        Thread.setDefaultUncaughtExceptionHandler(null);
        stopServices();
    }
}
```

8.3 有组织有纪律：线程工厂

从 JDK 1.5 开始，Java 标准库本身就支持创建线程的工厂方法（Factory Method）³。ThreadFactory 接口是工厂方法模式的一个实例，它定义了如下工厂方法：

```
public Thread newThread(Runnable r)
```

newThread 方法可以用来创建线程，该方法的参数 r 代表所创建的线程需要执行的任务。如果把线程对象看作某种“产品”，那么通过 new 方式创建线程就好比手工制作，而使用 ThreadFactory 接口创建线程则好比是工厂采用标准化的流水线进行生产。我们可以在 ThreadFactory.newThread 方法中封装线程创建的逻辑，这使得我们能够以统一的方式为线程的创建、配置做一些非常有用的动作。

在如清单 8-3 所示的例子中，ThreadFactory 实现类 XThreadFactory 的 newThread 方法为其创建的每一个线程做了这样一些列的处理逻辑：为线程关联 UncaughtExceptionHandler，为线程设置一个含义更加具体的有助于问题定位的名称，确保线程是一个用户线程，确保线程的优先级为正常级别，以及在线程创建的时候打印相关日志信息。并且，这些线程的 toString() 返回值更加有利于问题的定位——在对真实的（商用）多线程系统中的问题进行定位的过程中，将一个线程与另外一个线程区分开来非常有助于问题的定位，线程 ID 以及线程对象的身份标识（Hash Code）是将一个线程与另外一个线程区分开来的重要依据，而 Thread.toString() 的返回值并没有体现这一点。可见，XThreadFactory 不仅仅是为我们提供了一个新的线程，它还为此线程做了一些有利于简化客户端代码以及有利于代码调试和问题定位的动作。

清单 8-3 使用 ThreadFactory 创建线程

```
public class XThreadFactory implements ThreadFactory {
    final static Logger LOGGER = Logger.getAnonymousLogger();
    private final UncaughtExceptionHandler ueh;
    private final AtomicInteger threadNumber = new AtomicInteger(1);
    // 所创建的线程的线程名前缀
    private final String namePrefix;

    public XThreadFactory(UncaughtExceptionHandler ueh, String name) {
        this.ueh = ueh;
        this.namePrefix = name;
    }
    // ...
    public XThreadFactory() {
```

3 所谓工厂方法就是用于创建对象的方法。

```

        this(new LoggingUncaughtExceptionHandler(), "thread");
    }

    protected Thread doMakeThread(final Runnable r) {
        return new Thread(r) {
            @Override
            public String toString() {
                // 返回对问题定位更加有益的信息
                ThreadGroup group = getThreadGroup();
                String groupName = null == group ? "" : group.getName();
                String threadInfo = getClass().getSimpleName() + "[" + getName() + ","
                    + getId() + ","
                    + groupName + "]" + hashCode();
                return threadInfo;
            }
        };
    }

    @Override
    public Thread newThread(Runnable r) {
        Thread t = doMakeThread(r);
        t.setUncaughtExceptionHandler(ueh);
        t.setName(namePrefix + "-" + threadNumber.getAndIncrement());
        if (t.isDaemon()) {
            t.setDaemon(false);
        }
        if (t.getPriority() != Thread.NORM_PRIORITY) {
            t.setPriority(Thread.NORM_PRIORITY);
        }
        if (LOGGER.isLoggable(Level.FINE)) {
            LOGGER.fine("new thread created" + t);
        }
        return t;
    }

    static class LoggingUncaughtExceptionHandler implements
        UncaughtExceptionHandler {
        @Override
        public void uncaughtException(Thread t, Throwable e) {
            // 将线程异常终止的相关信息记录到日志中
            LOGGER.log(Level.SEVERE, t + " terminated:", e);
        }
    }
} // LoggingUncaughtExceptionHandler 类定义结束

```

8.4 线程的暂挂与恢复

`Thread.suspend()`、`Thread.resume()`这两个方法都是已废弃的方法。其作用分别是暂挂线程和恢复线程。暂挂（Suspend）与暂停的含义基本相同，它更多的是指用户（人）感知得到的线程暂停；恢复（Resume）与唤醒的含义基本相同，它更多的是指用户（人）感知得到的线程唤醒。我们可以采用与停止线程相似的思想来实现线程的暂挂与恢复：设置一个线程暂挂标志，线程每次执行比较耗时的操作前都先检查一下这个标志。如果该标志指示线程应该暂挂，那么线程就执行 `Object.wait()/Condition.await()` 暂停，直到其他线程重新设置暂挂标志并将其唤醒。根据该思路，我们可以设计一个用于控制线程的暂挂与恢复的工具类 `PauseControl`，如清单 8-4 所示。

清单 8-4 控制线程的暂挂与恢复的工具类

```
public class PauseControl extends ReentrantLock {
    private static final long serialVersionUID = 176912639934052187L;
    // 线程暂挂标志
    private volatile boolean suspended = false;
    private final Condition condSuspended = newCondition();

    /**
     * 暂停线程
     */
    public void requestPause() {
        suspended = true;
    }

    /**
     * 恢复线程
     */
    public void proceed() {
        lock();
        try {
            suspended = false;
            condSuspended.signalAll();
        } finally {
            unlock();
        }
    }

    /**
     * 当前线程仅在线程暂挂标记不为 true 的情况下才执行指定的目标动作
     *
     * @targetAction 目标动作
     * @throws InterruptedException
     */
}
```

```

*/
public void pauseIfNecessary(Runnable targetAction) throws InterruptedException
{
    lock();
    try {
        while (suspended) {
            condSuspended.await();
        }
        targetAction.run();
    } finally {
        unlock();
    }
}
}

```

PauseControl 本身继承自 ReentrantLock, 其 volatile 实例变量 suspended 充当线程暂挂标记。PauseControl.requestPause() 的作用仅仅是将 suspended 置为 true, 而 PauseControl.pauseIfNecessary() 则通过 Condition.await() 确保只有在 suspended 不为 true 的情况下指定的目标动作才会被执行。PauseControl.proceed() 的作用是将 suspended 置为 false 并唤醒所有被暂停的线程。PauseControl.requestPause()、PauseControl.proceed() 的作用分别相当于 Thread.suspend()、Thread.resume()。

清单 8-5 展示了一个利用 PauseControl 实现的线程的暂停与恢复的 Demo。该 Demo 模拟一个奴隶 (Slave) 干活时每隔一段时间询问其主人 (Master) 能否允许其休息一下。奴隶只有在得到主人允许的情况下才能够休息, 否则他必须继续干活! 这里, 奴隶停下手中的工作询问主人以及等到主人的许可后休息都是通过线程的暂挂来模拟的, 而休息过后继续干活则是通过线程的恢复来模拟的。

清单 8-5 暂停与恢复 Demo

```

public class ThreadPauseDemo {
    final static PauseControl pc = new PauseControl();

    public static void main(String[] args) {
        final Runnable action = new Runnable() {
            @Override
            public void run() {
                Debug.info("Master, I'm working...");
                Tools.randomPause(300);
            }
        };
        Thread slave = new Thread() {
            @Override
            public void run() {

```

```

        try {
            for (;;) {
                pc.pauseIfNecessary(action);
            }
        } catch (InterruptedException e) {
            // 什么也不做
        }
    }
}

};
slave.setDaemon(true);
slave.start();
askOnBehaveOfSlave();
}

static void askOnBehaveOfSlave() {
    String answer;
    int minPause = 2000;
    try (Scanner sc = new Scanner(System.in)) {
        for (;;) {
            Tools.randomPause(8000, minPause);
            pc.requestPause();
            Debug.info("Master, may I take a rest now?\n");
            Debug.info("\n(1) OK, you may take a rest\n"
                + "(2) No, Keep working!\nPress any other key to quit:\n");
            answer = sc.next();
            if ("1".equals(answer)) {
                pc.requestPause();
                Debug.info("Thank you, my master!");
                minPause = 8000;
            } else if ("2".equals(answer)) {
                Debug.info("Yes, my master!");
                pc.proceed();
                minPause = 2000;
            } else {
                break;
            }
        }
        // for 结束
    } // try 结束
    Debug.info("Game over!");
}
}

```

8.5 线程的高效利用：线程池

线程是一种昂贵的资源，其开销主要包括以下几个方面。

- 线程的创建与启动的开销。与普通的对象相比，Java 线程还占用了额外的存储空间——栈空间。并且，线程的启动会产生相应的线程调度开销。
- 线程的销毁。线程的销毁也有其开销。
- 线程调度的开销。线程的调度会导致上下文切换，从而增加处理器资源的消耗，使得应用程序本身可以使用的处理器资源减少。
- 一个系统能够创建的线程总是受限于该系统所拥有的处理器数目。无论是 CPU 密集型还是 I/O 密集型线程，这些线程的数量的临界值总是处理器的数目。

因此，从整个系统乃至整个主机的角度来看我们需要一种有效使用线程的方式。线程池就是有效使用线程的一种常见方式。

常见的对象池（比如数据库连接池）的实现方式是对象池（本身也是个对象）内部维护一定数量的对象，客户端代码需要一个对象的时候就向对象池申请（借用）一个对象，用完之后再将该对象返还给对象池，于是对象池中的一个对象就可以先后为多个客户端线程服务。线程池本身也是一个对象，不过它的实现方式与普通的对象池不同，如图 8-2 所示：线程池内部可以预先创建一定数量的工作者线程，客户端代码并不需要向线程池借用线程而是将其需要执行的任务作为一个对象提交给线程池，线程池可能将这些任务缓存在队列（工作队列）之中，而线程池内部的各个工作者线程则不断地从队列中取出任务并执行之。因此，线程池可以被看作基于生产者—消费者模式的一种服务，该服务内部维护的工作者线程相当于消费者线程，线程池的客户端线程相当于生产者线程，客户端代码提交给线程池的任务相当于“产品”，线程池内部用于缓存任务的队列相当于传输通道。

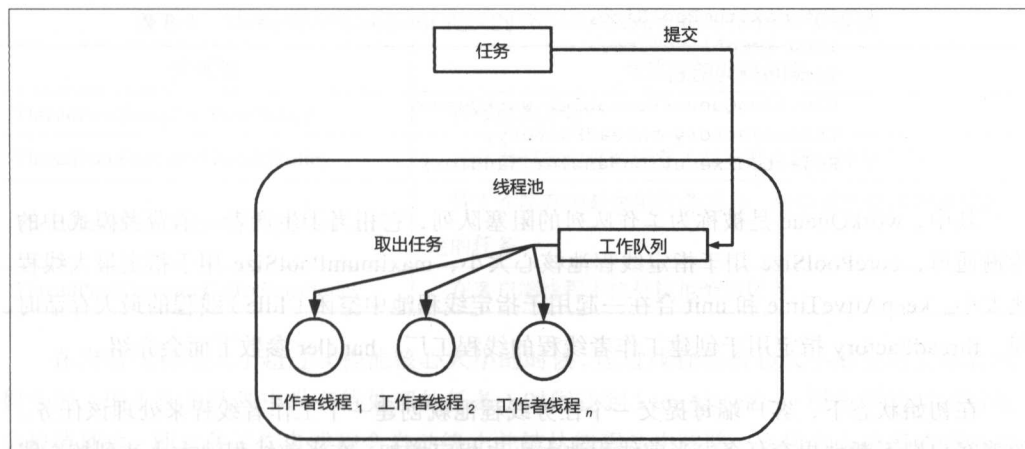


图 8-2 线程池原理示意图

`java.util.concurrent.ThreadPoolExecutor` 类就是一个线程池，客户端代码可以调用 `ThreadPoolExecutor.submit` 方法向其提交任务，`ThreadPoolExecutor.submit` 方法声明如下：

```
public Future<?> submit(Runnable task)
```

其中，`task` 参数是一个 `Runnable` 实例，它代表客户端需要线程池代为执行的任务。为便于讨论，这里我们先忽略该方法的返回值。

线程池内部维护的工作者线程的数量就被称为该线程池的线程池大小（Pool Size）。`ThreadPoolExecutor` 的线程池大小有 3 种形态：当前线程池大小（Current Pool Size）表示线程池中实际工作者线程的数量；最大线程池大小（Maximum Pool Size）表示线程池中允许存在的工作者线程的数量上限，其具体取值可参考第 4 章的式（4-5）；核心线程大小（Core Pool Size）表示一个不大于最大线程池大小的工作者线程数量上限。它们之间的数量关系如下：

当前线程池大小 \leq 核心线程池大小 \leq 最大线程池大小，

或核心线程池大小 \leq 当前线程池大小 \leq 最大线程池大小

这里，除了当前线程池大小是对线程池中现有的工作者线程进行计数的结果，其他有关线程池大小的概念实际上都是由开发人员或者系统配置数据指定的一个阈值（Threshold）。这些阈值的具体含义下文会介绍。

`ThreadPoolExecutor` 的构造器中包含参数数量最多的一个构造器的声明如下：

```
public ThreadPoolExecutor(int corePoolSize,
    int maximumPoolSize,
    long keepAliveTime,
    TimeUnit unit,
    BlockingQueue<Runnable> workQueue,
    ThreadFactory threadFactory,
    RejectedExecutionHandler handler)
```

其中，`workQueue` 是被称为工作队列的阻塞队列，它相当于生产者—消费者模式中的传输通道，`corePoolSize` 用于指定线程池核心大小，`maximumPoolSize` 用于指定最大线程池大小。`keepAliveTime` 和 `unit` 合在一起用于指定线程池中空闲（Idle）线程的最大存活时间。`threadFactory` 指定用于创建工作者线程的线程工厂。`handler` 参数下面会介绍。

在初始状态下，客户端每提交一个任务线程池就创建一个工作者线程来处理该任务。随着客户端不断地提交任务，当前线程池大小也相应增加。在当前线程池大小达到核心线程池大小的时候，新来的任务会被存入工作队列之中。这些缓存的任务由线程池中的所有

工作者线程负责取出进行执行。线程池将任务存入工作队列的时候调用的是 `BlockingQueue` 的非阻塞方法 `offer(E e)`，因此工作队列满并不会使提交任务的客户端线程暂停。当工作队列满的时候，线程池会继续创建新的工作者线程，直到当前线程池大小达到最大线程池大小。线程池是通过调用 `threadFactory.newThread` 方法来创建工作线程的。如果我们在创建线程池的时候没有指定线程工厂（即调用了 `ThreadPoolExecutor` 的其他构造器），那么 `ThreadPoolExecutor` 会使用 `Executors.defaultThreadFactory()` 所返回的默认线程工厂。当线程池饱和（Saturated）时，即工作者队列满并且当前线程池大小达到最大线程池大小的情况下，客户端试图提交的任务会被拒绝（Reject）。为了提高线程池的可靠性，Java 标准库引入了一个 `RejectedExecutionHandler` 接口用于封装被拒绝的任务的处理策略，该接口仅定义了如下方法：

```
void rejectedExecution(Runnable r, ThreadPoolExecutor executor)
```

其中，`r` 代表被拒绝的任务，`executor` 代表拒绝任务 `r` 的线程池实例。我们可以通过线程池的构造器参数 `handler` 或者线程池的 `setRejectedExecutionHandler` (`RejectedExecutionHandler handler`) 方法来为线程池关联一个 `RejectedExecutionHandler`。当客户端提交的任务被拒绝时，线程池所关联的 `RejectedExecutionHandler` 的 `rejectedExecution` 方法会被线程池调用。`ThreadPoolExecutor` 自身提供了几个现成的 `RejectedExecutionHandler` 接口实现类（见表 8-1），其中 `ThreadPoolExecutor.AbortPolicy` 是 `ThreadPoolExecutor` 使用的默认 `RejectedExecutionHandler`。如果默认的 `RejectedExecutionHandler`（它会直接抛出异常）无法满足要求，那么我们可以优先考虑 `ThreadPoolExecutor` 自身提供的其他 `RejectedExecutionHandler`，其次才去考虑使用自行实现的 `RejectedExecutionHandler` 接口。

表 8-1 `ThreadPoolExecutor` 提供的 `RejectedExecutionHandler` 实现类

实现类	所实现的处理策略
<code>ThreadPoolExecutor.AbortPolicy</code>	直接抛出异常
<code>ThreadPoolExecutor.DiscardPolicy</code>	丢弃当前被拒绝的任务（而不抛出任何异常）
<code>ThreadPoolExecutor.DiscardOldestPolicy</code>	将工作队列中最老的任务丢弃，然后重新尝试接纳被拒绝的任务
<code>ThreadPoolExecutor.CallerRunsPolicy</code>	在客户端线程中执行被拒绝的任务

在当前线程池大小超过线程池核心大小的时候，超过线程池核心大小部分的工作者线程空闲（即工作者队列中没有待处理的任务）时间达到 `keepAliveTime` 所指定的时间后就会被清理掉，即这些工作者线程会自动终止并被从线程池中移除。这种空闲线程清理机制有利于节约有限的线程资源，但是 `keepAliveTime` 值设置不合理（特别是设置得太小）可

能导致工作者线程频繁地被清理和创建反而增加了开销！

线程池中数量上等于核心线程池大小的那部分工作者线程，习惯上我们称之为核心线程（Core Thread）。如前文所述，当前线程池大小是随着线程池接收到的任务的数量而逐渐向核心线程池大小靠拢的，即核心线程是逐渐被创建与启动的。ThreadPoolExecutor.prestartAllCoreThreads()则使得我们可以使线程池在未接收到任何任务的情况下预先创建并启动所有核心线程，这样可以减少任务被线程池处理时所需的等待时间（等待核心线程的创建与启动）。

ThreadPoolExecutor.shutdown()/shutdownNow() 方法可用来关闭线程池。使用 shutdown()关闭线程池的时候，已提交的任务会被继续执行，而新提交的任务会像线程池饱和时那样被拒绝掉。ThreadPoolExecutor.shutdown()返回的时候线程池可能尚未关闭，即线程池中可能还有工作者线程正在执行任务。应用代码可以通过调用 ThreadPoolExecutor.awaitTermination(long timeout,TimeUnit unit)来等待线程池关闭结束。使用 ThreadPoolExecutor.shutdownNow()关闭线程池的时候，正在执行的任务会被停止，已提交而等待执行的任务也不会被执行。该方法的返回值是已提交而未被执行的任务列表，这为被取消的任务的重试提供了一个机会。由于 ThreadPoolExecutor.shutdownNow()内部是通过调用工作者线程的 interrupt 方法来停止正在执行的任务的，因此某些无法响应中断的任务可能永远也不会停止。反过来说，在关闭线程池的时候如果我们能够确保已经提交的任务都已执行完毕并且没有新的任务会被提交，那么调用 ThreadPoolExecutor.shutdownNow()总是安全可靠的。

在第4章第1个实战案例（大文件下载器）中，我们为每个下载子任务（DownloadTask 实例）都创建一个相应的工作者线程，虽然这样做也能够大幅提高下载效率，但是线程资源的利用可能并不高。另外，一个下载子任务执行失败意味着整个大文件下载的失败，因此一个工作者线程抛出异常的时候（为了简单起见，我们不对异常进行重试处理）其他工作者线程也就没有必要再运行下去而是要提前终止。为解决上述两个问题，我们可以使用 ThreadPoolExecutor 来改写 BigFileDownloader 类（参见清单 4-1）的 dispatchWork 方法和 doCleanup 方法（如清单 8-6 所示）。

清单 8-6 基于线程池的大文件下载器

```
public class TPBigFileDownloader extends BigFileDownloader {
    final static int N_CPU = Runtime.getRuntime().availableProcessors();
    final ThreadPoolExecutor executor = new ThreadPoolExecutor(2, N_CPU * 2, 4,
        TimeUnit.SECONDS,
        new ArrayBlockingQueue<Runnable>(N_CPU * 8),
        new ThreadPoolExecutor.CallerRunsPolicy());

    public TPBigFileDownloader(String file) throws Exception {
```

```

    super(file);
}

public static void main(String[] args) throws Exception {
    final int argc = args.length;
    TPBigFileDownloader downloader = new TPBigFileDownloader(args[0]);
    long reportInterval = argc >= 2 ? Integer.valueOf(args[1]) : 10;

    // 平均每个处理器执行 8 个下载子任务
    final int taskCount = N_CPU * 8;
    downloader.download(taskCount, reportInterval * 1000);
}

@Override
protected void dispatchWork(final DownloadTask dt, int workerIndex) {
    executor.submit(new Runnable() {
        @Override
        public void run() {
            try {
                dt.run();
            } catch (Exception e) {
                e.printStackTrace();
                // 任何一个下载子任务出现异常就取消整个下载任务
                cancelDownload();
            }
        }
    });
}

@Override
protected void doCleanup() {
    executor.shutdownNow();
    super.doCleanup();
}
}

```

这里，我们以实例变量 `executor` 的形式创建了一个线程池，用于负责文件下载子任务（`DownloadTask` 实例）的执行。该线程池的核心线程池大小为 2。考虑到该线程池的核心任务属于 I/O 密集型任务（参见第 4 章），因此我们将最大线程池大小设置为系统处理器数目的两倍。并且，我们使用 `ThreadPoolExecutor.CallerRunsPolicy` 作为线程池饱和处理策略，这意味着如果有下载子任务因线程池饱和而被拒绝，那么这些子任务将由 `dispatchWork` 方法的执行线程（即 `main` 线程）来执行，从而确保了程序的可靠性。`dispatchWork` 方法会为每个 `DownloadTask` 实例都创建一个包装任务，并在该包装中实现子任务下载异常处理逻辑，即在任何一个下载子任务处理失败的情况下都取消整个文件的下载。然后，

`dispatchWork` 方法将这个包装任务提交给线程池 `executor` 执行。在整个文件下载完毕（即下载进度为 100%）后 `doCleanup` 方法会被执行（由 `BigFileDownloader.download(int, long)` 调用）。由于 `doCleanup` 方法被执行的时候所有下载子任务都已经执行结束并且不会有新的子任务被提交，因此在 `doCleanup` 方法中我们可以调用 `executor.shutdownNow()` 来安全、可靠地将线程池关闭。通过对比上述程序与清单 4-1 中的程序的运行，我们可以发现使用线程池的方案与清单 4-1 所采用的直接使用工作者线程的方案相比在文件下载速率方面差别不大，但是前者所使用的线程数量要少得多，即提高了线程资源利用率。

从本案例可以看出，由于线程池（消费者）通常需要接收来自不同客户端（生产者）线程所提交的任务，因此一般情况下我们会以实例变量（或者静态变量）的形式来存储 `ThreadPoolExecutor` 实例。

8.5.1 任务的处理结果、异常处理与取消

在上述例子中，客户端代码（`TPBigFileDownloader`）仅向线程池提交任务（文件下载子任务）而不关心这些任务的处理结果数据。如果客户端关心任务的处理结果，那么它可以使用 `ThreadPoolExecutor` 的另外一个 `submit` 方法来提交任务，该 `submit` 方法的声明如下：

```
public <T> Future<T> submit(Callable<T> task)
```

`task` 参数代表客户端需要提交的任务，其类型为 `java.util.concurrent.Callable`。`Callable` 接口定义的唯一方法声明如下：

```
V call() throws Exception
```

`Callable` 接口也是对任务的抽象：任务的处理逻辑可以在 `Callable` 接口实现类的 `call` 方法中实现。`Callable` 接口相当于一个增强型的 `Runnable` 接口：`call` 方法的返回值代表相应任务的处理结果，其类型 `V` 是通过 `Callable` 接口的类型参数指定的；`call` 方法代表的任务在其执行过程中可以抛出异常。而 `Runnable` 接口中的 `run` 方法既无返回值也不能抛出异常。`Executors.callable(Runnable task, T result)` 能够将 `Runnable` 接口转换为 `Callable` 接口实例。

上述 `submit` 方法的返回值类型为 `java.util.concurrent.Future`。`Future` 接口实例可被看作提交给线程池执行的任务的处理结果句柄（Handle），`Future.get()` 方法可以用来获取 `task` 参数所指定的任务的处理结果，该方法声明如下：

```
V get() throws InterruptedException, ExecutionException
```

`Future.get()`被调用时,如果相应的任务尚未执行完毕,那么`Future.get()`会使当前线程暂停,直到相应的任务执行结束(包括正常结束和抛出异常而终止)。因此,`Future.get()`是个阻塞方法,该方法能够抛出`InterruptedException`说明它可以响应线程中断。另外,假设相应的任务执行过程中抛出一个任意的异常`originalException`,那么`Future.get()`方法本身就会抛出相应的`ExecutionException`异常。调用这个异常(`ExecutionException`)的`getCause()`方法可返回`originalException`。因此,客户端代码可以通过捕获`Future.get()`调用抛出的异常来知晓相应任务执行过程中抛出的异常。

由于在任务未执行完毕的情况下调用`Future.get()`方法来获取该任务的处理结果会导致等待并由此导致上下文切换,因此客户端代码应该尽可能早地向线程池提交任务,并尽可能晚地调用`Future.get()`方法来获取任务的处理结果,而线程池则正好利用这段时间来执行已提交的任务(包括我们关心的任务)。

注意 客户端代码应该尽可能早地向线程池提交任务,并仅在需要相应任务的处理结果数据的那一刻才调用`Future.get()`方法。

下面看一个`Future`接口使用的Demo。该Demo模拟从指定的车牌照片中识别出相应的车牌号(字符串)。这个识别的过程可能比较耗时,因此我们将这个识别任务封装为一个`Callable`实例提交给专门的线程池执行,并在需要该任务的处理结果数据(车牌号码)时才调用`Future.get()`,如清单8-7所示。

清单 8-7 获取线程池执行的任务处理结果 Demo

```
public class TaskResultRetrievalDemo {
    final static int N_CPU = Runtime.getRuntime().availableProcessors();
    final ThreadPoolExecutor executor = new ThreadPoolExecutor(0, N_CPU * 2, 4,
        TimeUnit.SECONDS,
        new ArrayBlockingQueue<Runnable>(100),
        new ThreadPoolExecutor.CallerRunsPolicy());

    public static void main(String[] args) {
        TaskResultRetrievalDemo demo = new TaskResultRetrievalDemo();
        Future<String> future = demo.recognizeImage("/tmp/images/0001.png");
        // 执行其他操作
        doSomething();
        try {
            // 仅在需要相应任务的处理结果时才调用 Future.get()
            Debug.info(future.get());
        } catch (InterruptedException e) {
            // 什么也不做
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
    }
}
```

```

    }
}

private static void doSomething() {
    Tools.randomPause(200);
}

public Future<String> recognizeImage(final String imageFile) {
    return executor.submit(new Callable<String>() {
        @Override
        public String call() throws Exception {
            return doRecognizeImage(new File(imageFile));
        }
    });
}

protected String doRecognizeImage(File imageFile) {
    String result = null;
    // 模拟实际运行结果
    String[] simulatedResults = { "苏 Z MM518", "苏 Z XYZ618", "苏 Z 007618" };
    result = simulatedResults[(int) (Math.random() * simulatedResults.length)];
    Tools.randomPause(100);
    // 省略其他代码
    return result;
}
}

```

Future 接口还支持任务的取消。为此，Future 接口定义了如下方法：

```
boolean cancel(boolean mayInterruptIfRunning)
```

该方法的返回值表示相应的任务取消是否成功。任务取消失败的原因包括待取消的任务已执行完毕或者正在执行、已经被取消以及其他无法取消因素。参数 `mayInterruptIfRunning` 表示是否允许通过给相应任务的执行线程发送中断来取消任务。`Future.isCancelled()` 返回值代表相应的任务是否被成功取消。由于一个任务被成功取消之后，相应的 `Future.get()` 调用会抛出 `CancellationException` 异常（运行时异常），因此如果任务有可能会被取消，那么在获取任务的处理结果之前，我们需要先判断任务是否已经被取消了。

`Future.isDone()` 方法可以检测相应的任务是否执行完毕。任务执行完毕、执行过程中抛出异常以及任务被取消都会导致该方法返回 `true`。

`Future.get()` 会使其执行线程无限制地等待，直到相应的任务执行结束。商用系统中这种无时间限制的等待往往是不现实的。此时我们可以使用 `get` 方法的另外一个版本，其声

明如下：

```
V get(long timeout, TimeUnit unit) throws InterruptedException,
ExecutionException, TimeoutException
```

该方法的作用与 `Future.get()` 相同，不过它允许我们指定一个等待超时时间。如果在该时间内相应的任务未执行结束，那么该方法就会抛出 `TimeoutException`。由于该方法参数中指定的超时时间仅仅用于控制客户端线程（即该方法的执行线程）等待相应任务的处理结果最多会等待多长时间，而非相应任务本身的执行时间限制，因此，客户端线程通常需要在捕获 `TimeoutException` 之后执行 `Future.cancel(true)` 来取消相应任务的执行（因为此时我们已经不再需要该任务的处理结果了）。

8.5.2 线程池监控

尽管线程池的大小、工作队列的容量、线程空闲时间限制这些线程池的属性可通过配置的方式进行指定（而不是硬编码在代码中），但是所指定的值是否恰当则需要通过监控来判断。例如，如果我们选择有界队列作为工作队列，那么这个队列的容量以多少为宜呢，这需要在软件测试过程中对线程池进行监控来确定。另外，考虑到测试环境和软件实际运行环境总是存在差别的，出于软件运维的考虑我们也可能需要对线程池进行监控。`ThreadPoolExecutor` 类提供了对线程池进行监控的相关方法，如表 8-2 所示。

表 8-2 `ThreadPoolExecutor` 提供的线程池监控相关方法

方 法	用 途
<code>getPoolSize()</code>	获取当前线程池大小
<code>getQueue()</code>	返回工作队列实例，通过该实例可获取工作队列的当前大小
<code>getLargestPoolSize()</code>	获取工作者线程数曾经达到的最大数，该数值有助于确认线程池的最大大小设置是否合理
<code>getActiveCount()</code>	获取线程池中当前正在执行任务的工作者线程数（近似值）
<code>getTaskCount()</code>	获取线程池到目前为止所接收到的任务数（近似值）
<code>getCompletedTaskCount()</code>	获取线程池到目前为止已经处理完毕的任务数（近似值）

此外，`ThreadPoolExecutor` 提供的两个钩子方法（Hook Method）：`beforeExecute(Thread t, Runnable r)` 和 `afterExecute(Thread t, Runnable r)` 也能够用于实现监控。设 `executor` 为任意一个 `ThreadPoolExecutor` 实例，在任意一个任务 `r` 被线程池 `executor` 中的任意一个工作者线程 `t` 执行前，`executor.beforeExecute(t,r)` 会被执行；当 `t` 执行完 `r` 之后，不管 `r` 的执行是否是成功的还是抛出了异常，`executor.afterExecute(t,r)` 始终会被执行。因此，如果有必要

的话我们可以通过创建 `ThreadPoolExecutor` 的子类并在子类的 `beforeExecute/afterExecute` 方法实现监控逻辑，比如计算任务执行的平均耗时。

8.5.3 线程池死锁

如果线程池中执行的任务在其执行过程中又会向同一个线程池提交另外一个任务，而前一个任务的执行结束又依赖于后一个任务的执行结果，那么就有可能出现这样的情形：线程池中的所有工作者线程都处于等待其他任务的处理结果而这些任务仍在工作队列中等待执行，这时由于线程池中已经没有可以对工作队列中的任务进行处理的工作者线程，这种等待就会一直持续下去从而形成死锁（`Deadlock`）。

因此，适合提交给同一线程池实例执行的任务是相互独立的任务，而不是彼此有依赖关系的任务。对于彼此存在依赖关系的任务，我们可以考虑分别使用不同的线程池实例来执行这些任务。

注意 同一个线程池只能用于执行相互独立的任务。彼此有依赖关系的任务需要提交给不同的线程池执行以避免死锁。

8.5.4 工作者线程的异常终止

如果任务是通过 `ThreadPoolExecutor.submit` 调用提交给线程池的，那么这些任务在其执行过程中即便是抛出了未捕获的异常也不会导致对其进行执行的工作者线程异常终止。当然，上文我们已经介绍过这种情形下任务所抛出的异常可以通过 `Future.get()` 所抛出的 `ExecutionException` 来获取。

如果任务是通过 `ThreadPoolExecutor.execute` 方法提交给线程池的，那么这些任务在其执行过程中一旦抛出了未捕获的异常，则对其进行执行的工作者线程就会异常终止。尽管 `ThreadPoolExecutor` 能够侦测到这种情况并在工作者线程异常终止的时候创建并启动新的替代工作者线程，但是由于线程的创建与启动都有其开销，因此这种情形下我们会尽量避免任务在其执行过程中抛出未捕获的异常。我们可以通过 `ThreadPoolExecutor` 的构造器参数或者 `ThreadPoolExecutor.setThreadFactory` 方法为线程池关联一个线程工厂。在这个线程工厂里面我们可以为其创建的工作者线程关联一个 `UncaughtExceptionHandler`，通过这个关联的 `UncaughtExceptionHandler` 我们可以侦测到任务执行过程中抛出的未捕获异常。不过，由于 `ThreadPoolExecutor` 内部实现的原因，只有通过 `ThreadPoolExecutor.execute` 调用（而不是 `ThreadPoolExecutor.submit` 调用）提交给线程池执行的任务，其执行过程中抛出

的未捕获异常才会导致 `UncaughtExceptionHandler.uncaughtException` 方法被调用。

注意

通过 `ThreadPoolExecutor.submit` 调用提交给线程池执行的任务，其执行过程中抛出的未捕获异常并不会导致与该线程池中的工作者线程关联的 `UncaughtExceptionHandler` 的 `uncaughtException` 方法被调用。

8.6 本章小结

本章介绍了如何将线程管控起来以便高效、可靠地利用线程这种有限的资源。本章知识结构如图 8-3 所示。

线程组是 `Thread.UncaughtExceptionHandler` 的一个实现类，它可以帮助我们检测线程的异常终止。多数情况下，我们可以忽略线程组这一概念以及线程组的存在。

`Thread.UncaughtExceptionHandler` 接口使得我们能够侦测到线程运行过程中抛出的未捕获的异常，以便做出相应的补救措施，例如创建并启动相应的替代线程。一个线程在其抛出未捕获的异常而终止前，总有一个 `UncaughtExceptionHandler` 实例会被选中。被选中的 `UncaughtExceptionHandler` 实例的 `uncaughtException` 方法会被该线程在其终止前执行。`UncaughtExceptionHandler` 实例选择的优先级：线程实例本身关联的 `UncaughtExceptionHandler` 实例 > 线程所在线程组 > 默认 `UncaughtExceptionHandler`。

线程工厂 `ThreadFactory` 能够封装线程的创建与配置的逻辑，这使得我们能够对线程的创建与配置进行统一的控制。

利用条件变量我们能够实现线程的暂挂与恢复，用于替代 `Thread.suspend()/resume()` 这两个废弃的方法。

线程池是生产者—消费者模式的一个具体例子，它能够摊销线程的创建、启动与销毁的开销，并在一定程度上有利于减少线程调动的开销。线程池使得我们能够充分利用有限的线程资源。`ThreadPoolExecutor` 支持核心线程大小以及最大线程池大小这两种阈值来控制线程池中的工作者线程总数。`ThreadPoolExecutor` 支持对核心线程以外的空闲了指定时间的工作者线程进行清理，以减少不必要的资源消耗。`RejectedExecutionHandler` 接口使得我们能够对被线程池拒绝的任务进行重试以提高系统的可靠性。`Future` 接口使得我们可以获取提交给线程池执行的任务的处理结果、侦测任务处理异常以及取消任务的执行。当一个线程池实例不再被需要的时候，我们需要主动将其关闭以节约资源。`ThreadPoolExecutor` 提供了一组能够对线程池进行监控的方法，通过这些方法我们能够了解线程池的当前线程

池大小、工作队列的情况等数据。同一个线程池只能用于执行相互独立的任务，彼此有依赖关系的任务需要提交给不同的线程池执行以避免死锁。我们可以通过线程工厂为线程池中的工作者线程关联 `UncaughtExceptionHandler`，但是这些 `UncaughtExceptionHandler` 只会对通过 `ThreadPoolExecutor.execute` 方法提交给线程池的任务起作用。

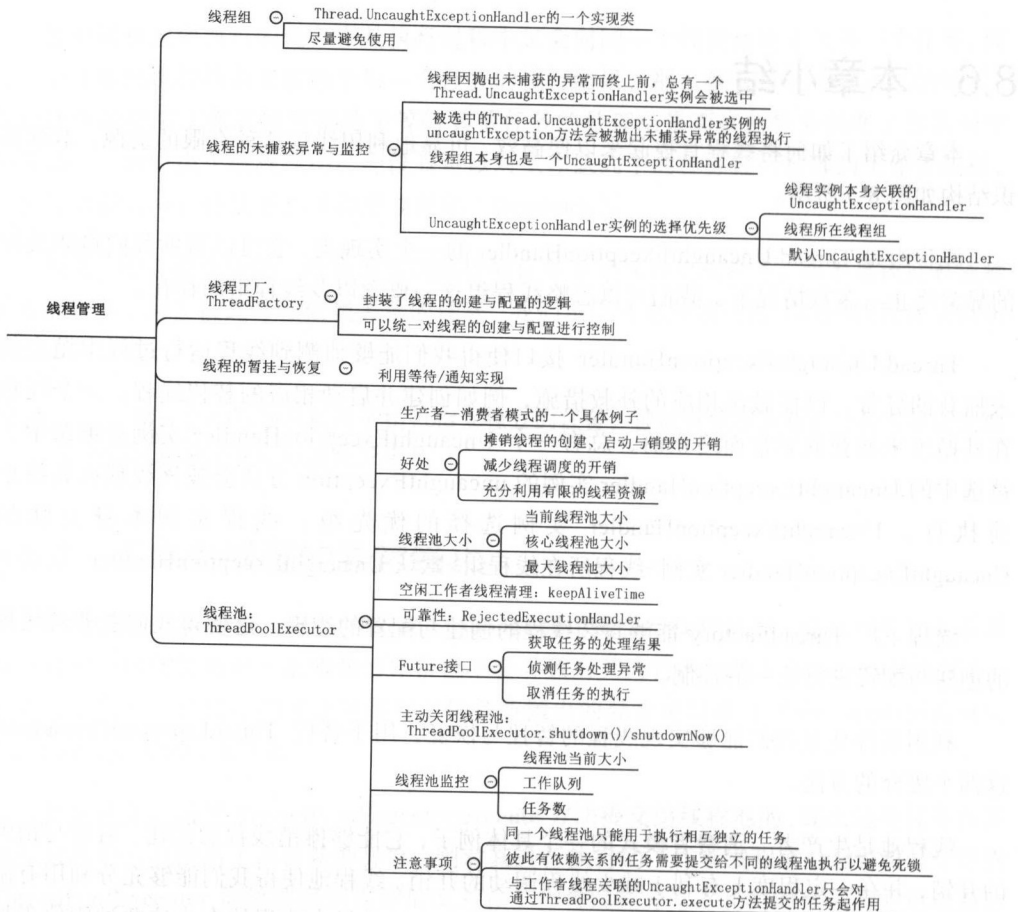


图 8-3 本章知识结构图

Java 异步编程

本章介绍了异步计算的概念以及 Java 标准库所提供的异步计算相关 API。

9.1 同步计算与异步计算

从多个任务的角度来看，任务可以是串行执行的，也可以是并发执行的。从单个任务的角度来看，任务的执行方式可以是同步的（Synchronous），如图 9-1（a）所示；也可以是异步的（Asynchronous），如图 9-1（b）所示。这里的同步与线程同步机制中的“同步”不是同一个概念。

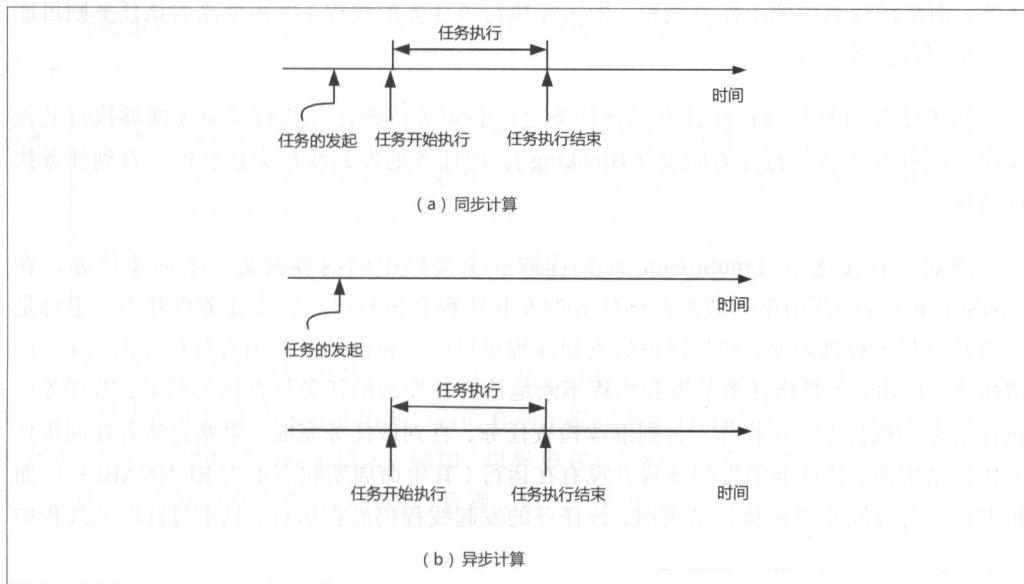


图 9-1 同步、异步计算示意图

以同步方式执行的任务，我们称之为同步任务，其任务的发起与任务的执行是在同一条时间线上进行的。换言之，任务的发起与任务的执行是串行的。同步任务就好比我们以电话的形式将一个消息通知给朋友的情形：我们先拨打对方的号码（任务的发起），只有在电话接通（任务开始执行）之后我们才能够将消息告诉对方（任务执行的过程）¹。

以异步方式执行的任务，我们称之为异步任务，其任务的发起与任务的执行是在不同的时间线上进行的。换言之，任务的发起与任务的执行是并发的。异步任务好比我们以短信的形式将一个消息通知给朋友的情形：我们只要给对方发送一条短信（任务的发起）便认为已经通知到对方了，而不必关心对方何时阅读这条短信，而实际上对方可能在第二天阅读这条短信（任务开始执行）。

同步方式与异步方式的说法是相对的：同一个任务我们既可以说它是异步任务，也可以说它是同步任务。假设我们用一个 `Runnable` 实例 `task` 来表示一个任务，如果我们直接调用 `task.run()` 来执行该任务，那么我们就可以称该任务为同步任务；如果我们通过 `new Thread(task).start()` 调用创建并启动一个专门的工作者线程来执行该任务，或者将该任务提交给一个 `Executor` 实例 `executor` 执行（即调用 `executor.execute(task)`），那么我们就可以称该任务为异步任务。同步方式与异步方式的称呼不仅仅取决于一个任务的具体执行方式，还取决于我们的观察角度。在上述例子中，假设我们将 `task` 提交给线程池执行，那么从该任务提交线程（即 `ThreadPoolExecutor.submit` 方法的执行线程）的角度来看它是一个异步任务，而从线程池中的工作者线程（即实际执行该任务的线程）的角度来看该任务则可能是一个同步任务²。

同步任务的发起线程在其发起该任务之后必须等待该任务执行结束才能够执行其他操作，这种等待往往意味着阻塞（`Blocking`），即任务的发起线程会被暂停，直到任务执行结束。

例如，直接通过 `InputStream.read()` 读取一个文件中的内容就是一个同步任务，在 `InputStream.read()` 调用返回数据前该任务的发起线程会被暂停。同步任务也并不一定总是会使其发起线程被阻塞，同步任务的发起线程也可能以轮询的方式来等待任务的结束。所谓轮询（`Polling`）是指任务的发起线程不断地检查其发起的任务是否执行结束，若任务已执行结束则执行下一步操作，否则继续检查任务，直到该任务完成。阻塞意味着在同步任务执行结束前，该任务的发起线程并没有在运行（其生命周期状态不为 `RUNNABLE`），而轮询意味着在同步任务执行结束前，该任务的发起线程仍然在运行，只不过此时该线程的

1 这里我们假设对方没有开通语音留言的功能。

2 当然，从线程池的角度来看，该任务可能仍然还是一个异步任务，这取决于这个任务的具体实现代码。

主要动作是检查相应的任务是否执行结束。同步任务的发起线程是采用阻塞的方式还是轮询方式来等待任务的结束很大程度上取决于我们使用的 API。例如,使用 `java.nio.channels.Selector` 类来编写网络应用程序的服务端代码的时候,我们能够采用轮询的方式来实现等待同步任务的结束,而多数情况下我们只能以阻塞方式来实现等待同步任务的结束。单个线程便可以实现同步任务的执行。在使用单个线程的情况下,多个同步任务只能以同步的方式执行。

异步任务的发起线程在其发起该任务之后不必等待该任务结束便可以继续执行其他操作,即异步任务的发起与实际执行可以是并发的。多线程编程本质上是异步的。比如一个线程通过 `ThreadPoolExecutor.submit(Callable<T>)` 调用向线程池提交一个任务(任务的发起),在该调用返回之后该线程便可以执行其他操作了,而该任务可能在此之后才被线程池中的某一个工作者线程所执行,这里任务的提交与执行是并发的,而不是串行的。可见,异步任务可以使其发起线程不必因等待其执行结束而被阻塞,即异步任务执行方式往往意味着非阻塞(Non-blocking)。然而,阻塞与非阻塞只是任务执行方式的一种属性,它与任务执行方式之间并没有必然的关系:同步任务执行方式多数情况下意味着阻塞,但是它也可能意味着非阻塞(轮询);异步任务执行方式多数情况下意味着非阻塞,但是它也可能意味着阻塞。例如,如果我们在向线程池提交一个任务之后立刻调用 `Future.get()` 来试图获取该任务的处理结果(即 `ThreadPoolExecutor.submit(someTask).get()`),那么尽管该任务是异步执行的,但是其发起线程仍然可能由于 `Future.get()` 调用时该任务尚未被线程池执行结束而被阻塞。异步任务的执行需要借助多个线程来实现。多个异步任务能够以并发的方式被执行。

注意

- 阻塞与非阻塞只是任务执行方式(同步/异步)本身的一种属性,它们与任务执行方式之间并未有必然的联系:异步任务既可能是非阻塞的,也可能是阻塞的;同步任务既可能是阻塞的,也可能是非阻塞的。
- 同步方式与异步方式的说法是相对的,它取决于任务的执行方式以及我们的观察角度。

同步方式的优点是代码简单、直观,缺点是它往往意味着阻塞,而阻塞会限制系统的吞吐率。异步方式往往意味着非阻塞,因而有利于提高系统的吞吐率。异步方式的代价是更为复杂的代码和更多的资源投入。例如,以异步方式执行任务需要借助额外的工作者线程,并且还需要对这些工作者线程进行管理(启动、停止等)。

9.2 Java Executor 框架

`Runnable` 接口和 `Callable` 接口都是对任务处理逻辑的抽象，这种抽象使得我们无须关心任务的具体处理逻辑：不管是什么样的任务，其处理逻辑总是展现为一个具有统一签名的方法——`Runnable.run()` 或者 `Callable.call()`。`java.util.concurrent.Executor` 接口则是对任务的执行进行的抽象，该接口仅定义了如下方法：

```
void execute(Runnable command)
```

其中，`command` 参数代表需要执行的任务。`Executor` 接口使得任务的提交方（相当于生产者）只需要知道它调用 `Executor.execute` 方法便可以使指定的任务被执行，而无须关心任务具体的执行细节：比如，任务是采用一个专门的工作者线程执行的，还是采用线程池执行的；采用什么样的线程池执行的；多个任务是以何种顺序被执行的。可见，`Executor` 接口使得任务的提交能够与任务执行的具体细节解耦（Decoupling）。和对任务处理逻辑的抽象类似，对任务执行的抽象也能给我们带来信息隐藏（Information）和关注点分离（Separation Of Concern）的好处。

解耦任务的提交与任务的具体执行细节所带来的好处的一个例子是，它在一定程度上能够屏蔽任务同步执行与异步执行的差异。例如，对于同一个任务（`Runnable` 实例），如果我们把它提交给一个 `ThreadPoolExecutor`（它实现了 `Executor` 接口）执行，那么该任务就是异步执行；如果把这个任务提交给如清单 9-1 所示的 `Executor` 实例执行，那么该任务就是同步执行。这个任务不管是同步执行还是异步执行，对于其提交方来说并没有太大差别，这就为更改任务的具体执行方式提供了灵活性和便利；更改任务的具体执行细节可能不会影响到任务的提交方，而这意味着更小的代码改动量和测试量。

清单 9-1 使用 `Executor` 接口实现任务的同步执行

```
public class SynchronousExecutor implements Executor {
    @Override
    public void execute(Runnable command) {
        command.run();
    }
}
```

可见，`Executor` 接口一定程度上缩小了同步编程与异步编程的代码编写方式。

`Executor` 接口比较简单，功能也十分有限：首先，它只能为客户端代码执行任务，而无法将任务的处理结果返回给客户端代码；其次，`Executor` 接口实现类内部往往会维护一些工作者线程，当我们不再需要一个 `Executor` 实例的时候，往往需要主动将该实例内部维

护的工作者线程停掉以释放相应的资源，而 `Executor` 接口并没有定义相应的方法。

`ExecutorService` 接口继承自 `Executor` 接口，它解决了上述问题。`ExecutorService` 接口定义了几个 `submit` 方法，这些方法能够接受 `Callable` 接口或者 `Runnable` 接口表示的任务并返回相应的 `Future` 实例，从而使客户端代码提交任务后可以获取任务的执行结果。`ExecutorService` 接口还定义了 `shutdown()` 方法和 `shutdownNow()` 方法来关闭相应的服务(比如关闭其维护的工作者线程)。`ThreadPoolExecutor` 是 `ExecutorService` 的默认实现类。

9.2.1 实用工具类 Executors

第 8 章我们已经介绍到实用工具类 `java.util.concurrent.Executors`，它除了能够返回默认线程工厂 (`Executors.defaultThreadFactory()`)、能够将 `Runnable` 实例转换为 `Callable` 实例 (`Executors.callable` 方法) 之外，还提供了一些能够返回 `ExecutorService` 实例的快捷方法，如表 9-1 所示。这些 `ExecutorService` 实例往往使我们在不必手动创建 `ThreadPoolExecutor` 实例的情况下使用线程池。

表 9-1 Executors 提供的能够返回 ExecutorService 实例的快捷方法

方 法	适用条件及注意实现
<code>public static ExecutorService newCachedThreadPool()</code>	适用于执行大量耗时较短且提交比较频繁的任务。如果提交的任务执行耗时较长，那么可能导致线程池中的工作者线程无限地增加，最后导致过多的上下文切换，从而使得整个系统变慢
<code>public static ExecutorService newCachedThreadPool(ThreadFactory threadFactory)</code>	
<code>public static ExecutorService newFixedThreadPool(int nThreads)</code>	由于该方法返回的线程池的核心线程池大小等于其最大线程池大小，因此该线程池中的工作者线程永远不会超时。我们必须在不再需要该线程池时主动将其关闭
<code>public static ExecutorService newFixedThreadPool(int nThreads, ThreadFactory threadFactory)</code>	
<code>public static ExecutorService newSingleThreadExecutor()</code>	适合用来实现单(多)生产者—单消费者模式。该方法的返回值无法被转换为 <code>ThreadPoolExecutor</code> 类型
<code>public static ExecutorService newSingleThreadExecutor(ThreadFactory threadFactory)</code>	

- `Executors.newCachedThreadPool()`。该方法的返回值相当于：

```
new ThreadPoolExecutor(0, Integer.MAX_VALUE, 60L, TimeUnit.SECONDS,  
new SynchronousQueue<Runnable>());
```


即一个核心线程池大小为 0，最大线程池大小不受限，工作者线程允许的最大空闲时间（keepAliveTime）为 60 秒，内部以 SynchronousQueue 为工作队列（以下称之为 workerQueue）的一个线程池。这种配置意味着该线程池中的所有工作者线程在空闲了指定的时间后都可以被自动清理掉。由于该线程池的核心线程池大小为 0，因此提交给该线程池执行的第一个任务会导致该线程池中的第一个工作者线程被创建并启动。后续继续给该线程池提交任务的时候，由于当前线程池大小已经超过核心线程池大小（0），因此 ThreadPoolExecutor 此时会将任务缓存到工作队列之中（即调用 workerQueue.offer 方法）。

SynchronousQueue 内部并不维护用于存储队列元素的实际存储空间。一个线程（生产者线程）在执行 SynchronousQueue.offer(E) 的时候，如果没有其他线程（消费者线程）因执行 SynchronousQueue.take() 而被暂停，那么 SynchronousQueue.offer(E) 调用会直接返回 false，即入队列失败。因此，在该线程池中的所有工作者线程都在执行任务，即无空闲工作者线程的情况下给其提交任务会导致该任务无法被缓存成功。而 ThreadPoolExecutor 在任务缓存失败且线程池当前大小未达到最大线程池大小（这里的最大线程池大小实际上相当于无限）的情况下会创建并启动新的工作者线程。在极端的情况下，给该线程池每提交一个任务都会导致一个新的工作者线程被创建并启动，而这最终会导致系统中的线程过多，从而导致过多的上下文切换而使得整个系统被拖慢。因此，Executors.newCachedThreadPool() 所返回的线程池适合于用来执行大量耗时较短且提交频率较高的任务。而提交频率较高且耗时较长的任务（尤其是包含阻塞操作的任务）则不适合用 Executors.newCachedThreadPool() 所返回的线程池来执行。

- Executors.newFixedThreadPool(int nThreads)。该方法的返回值相当于：

```
new ThreadPoolExecutor(nThreads, nThreads, 0L, TimeUnit.MILLISECONDS,
    new LinkedBlockingQueue<Runnable>());
```

即一个以无界队列为工作队列，核心线程池大小与最大线程池大小均为 nThreads 且线程池中的空闲工作者线程不会被自动清理的线程池，这是一种线程池大小一旦达到其核心线程池大小就既不会增加也不会减少工作者线程的固定大小的线程池。因此，这样的线程池实例一旦不再需要，我们必须主动将其关闭。

- Executors.newSingleThreadExecutor()。该方法的返回值基本相当于 Executors.newFixedThreadPool(1) 所返回的线程池。不过，该线程池并非 ThreadPoolExecutor 实例，而是一个封装了 ThreadPoolExecutor 实例且对外仅暴露 ExecutorService 接口所定义的方法的一个 ExecutorService 实例。该线程池便于我们实现单（多）生

生产者-单消费者模式。该线程池确保了在任意一个时刻只有一个任务会被执行，这就形成了类似锁将原本并发的操作改为串行的操作的效果。因此，该线程池适合于用来执行访问了非线性安全对象而我们又不希望因此而引入锁的任务。该线程池也适合于用来执行 I/O 操作，因为 I/O 操作往往受限于相应的 I/O 设备，使用多个线程执行同一种 I/O 操作（比如多个线程各自读取一个文件）可能并不会提高 I/O 效率，所以如果使用一个线程执行 I/O 足以满足要求，那么仅使用一个线程即可，这样可以保障程序的简单性以避免一些不必要的问题（比如死锁）。

9.2.2 异步任务的批量执行：CompletionService

尽管 Future 接口使得我们能够方便地获取异步任务的处理结果，但是如果需要一次性提交一批异步任务并获取这些任务的处理结果的话，那么仅使用 Future 接口写出来的代码将颇为烦琐。java.util.concurrent.CompletionService 接口为异步任务的批量提交以及获取这些任务的处理结果提供了便利。

CompletionService 接口定义的一个 submit 方法可用于提交异步任务，该方法的签名与 ThreadPoolExecutor 的一个 submit 方法相同：

```
Future<V> submit(Callable<V> task)
```

task 参数代表待执行的异步任务，该方法的返回值可用于获取相应异步任务的处理结果。如果是批量提交异步任务，那么通常我们并不关心该方法的返回值。若要获取批量提交的异步任务的处理结果，那么我们可以使用 CompletionService 接口专门为此定义的方法，其中的一个方法是：

```
Future<V> take() throws InterruptedException
```

该方法与 BlockingQueue.take() 相似，它是一个阻塞方法，其返回值是一个已经执行结束的异步任务对应的 Future 实例，该实例就是提交相应任务时 submit(Callable<V>) 调用的返回值。如果 take() 被调用时没有已执行结束的异步任务，那么 take() 的执行线程就会被暂停，直到有异步任务执行结束。因此，我们批量提交了多少个异步任务，则多少次连续调用 CompletionService.take() 便可以获取这些任务的处理结果。

CompletionService 也定义了两个非阻塞方法用于获取异步任务的处理结果：

```
Future<V> poll()
```

```
Future<V> poll(long timeout, TimeUnit unit) throws InterruptedException
```

这两个方法与 BlockingQueue 的 poll 方法相似，它们的返回值是已执行结束的异步任

务对应的 Future 实例。

Java 标准库提供的 CompletionService 接口的实现类是 ExecutorCompletionService。ExecutorCompletionService 的一个构造器是：

```
ExecutorCompletionService(Executor executor,
                           BlockingQueue<Future<V>> completionQueue)
```

由此可见，ExecutorCompletionService 相当于 Executor 实例与 BlockingQueue 实例的一个融合体。其中，Executor 实例负责接收并执行异步任务，而 BlockingQueue 实例则用于存储已执行完毕的异步任务对应的 Future 实例。ExecutorCompletionService 会为其客户端提交的每个异步任务（Callable 实例或者 Runnable 实例）都创建一个相应的 Future 实例，通过该实例其客户端代码便可以获取相应异步任务的处理结果。ExecutorCompletionService 每执行完一个异步任务，就将该任务对应的 Future 实例存入其内部维护的 BlockingQueue 实例之中，而其客户端代码则可以通过 ExecutorCompletionService.take() 调用来获取这个 Future 实例。

使用 ExecutorCompletionService 的另外一个构造器 ExecutorCompletionService(Executor executor) 创建实例相当于：

```
new ExecutorCompletionService<V>(executor,
                                   new LinkedBlockingQueue<Future<V>> ());
```

下面看一个实战案例，该案例中我们使用 ExecutorCompletionService 以异步方式实现文件的批量 FTP 上传，如清单 9-2 所示。FileBatchUploader.uploadFiles 方法能够将指定的一批文件以异步方式上传到指定的 FTP 服务器³：该方法将这批文件的上传视作一个任务（以下称之为原始任务）并创建一个相应的 Runnable 实例将其提交给 dispatcher（Executor 实例）执行。原始任务的执行是通过调用 doUploadFiles 方法实现的。在 doUploadFiles 方法中，我们为原始任务中的每个文件都创建一个相应的文件上传任务（UploadTask 实例），并将这些任务批量提交给 completionService（CompletionService 实例）执行。然后，对于原始任务中的每个文件，一旦一个文件上传结束，即 completionService.take() 调用返回，那么我们就将该文件移动到备份目录并为该文件生成相应的 MD5 摘要文件⁴。接着，我们为每个 MD5 摘要文件创建一个相应的文件上传任务，并将其提交给 completionService 执

3 FTP 服务器由构造器中的 ftpServer 参数指定。

4 这种文件中的内容为相应文件对应的 MD5 摘要值。这里，MD5 摘要文件的作用一方面是供对方（即使用上传的文件的程序）进行数据完整性校验，另一方面它充当了原始任务中相应文件上传完毕的标记，即对方只有在“看到”一个 MD5 文件的情况下才能认为相应的原始任务文件的上传是结束的。

行。这里，对于原始任务中的每个文件，文件的实际上传是在一个线程（即 `Executor` 实例 `es` 中维护的一个单工作者线程）中执行的，而在该文件上传完毕后将其移动到备份目录以及生成相应的 MD5 文件这些操作则是在另外一个线程（即 `Executor` 实例 `dispatcher` 中维护的一个单工作者线程）中执行的，即文件的上传与对上传完毕文件的后续处理是并发的。这种并发得以实现正是得益于 `CompletionService` 所支持的批量异步任务提交以及获取执行任务对应的 `Future` 实例。

清单 9-2 使用 `ExecutorCompletionService` 实现文件异步批量上传

```
public class FileBatchUploader implements Closeable {
    // 完整代码见本书配套下载资源
    private final CompletionService<File> completionService;
    private final ExecutorService es;
    private final ExecutorService dispatcher;

    public FileBatchUploader(String ftpServer, String userName, String password,
        String targetRemoteDir) {
        // 完整代码见本书配套下载资源
        // 使用单工作者线程的线程池
        this.es = Executors.newSingleThreadExecutor();
        this.dispatcher = Executors.newSingleThreadExecutor();
        this.completionService = new ExecutorCompletionService<File>(es);
    }

    public void uploadFiles(final Set<File> files) {
        dispatcher.submit(new Runnable() {
            @Override
            public void run() {
                try {
                    doUploadFiles(files);
                } catch (InterruptedException ignored) {
                }
            }
        });
    }

    private void doUploadFiles(Set<File> files) throws InterruptedException {
        // 批量提交文件上传任务
        for (final File file : files) {
            completionService.submit(new UploadTask(file));
        }

        Future<File> future;
        File md5File;
        File uploadedFile;
        Set<File> md5Files = new HashSet<File>();
    }
}
```

```

for (File file : files) {
    try {
        future = completionService.take();
        uploadedFile = future.get();
        // 将上传成功的文件移动到备份目录，并为其生成相应的 MD5 文件
        md5File = generateMD5(moveToSuccessDir(uploadedFile));
        md5Files.add(md5File);
    } catch (ExecutionException | IOException | NoSuchAlgorithmException e) {
        e.printStackTrace();
        moveToDeadDir(file);
    }
}

for (File file : md5Files) {
    // 上传相应的 MD5 文件
    completionService.submit(new UploadTask(file));
}

// 检查 MD5 文件的上传结果
int successUploaded = md5Files.size();
for (int i = 0; i < successUploaded; i++) {
    future = completionService.take();
    try {
        uploadedFile = future.get();
        md5Files.remove(uploadedFile);
    } catch (ExecutionException e) {
        e.printStackTrace();
    }
}

// 将剩余（即未上传成功）的 MD5 文件移动到相应的备份目录
for (File file : md5Files) {
    moveToDeadDir(file);
}
}

private File generateMD5(File file) throws IOException, NoSuchAlgorithmException {
    String md5 = Tools.md5sum(file);
    File md5File = new File(file.getAbsolutePath() + ".md5");
    Files.write(Paths.get(md5File.getAbsolutePath()), md5.getBytes("UTF-8"));
    return md5File;
}

private static File moveToSuccessDir(File file) {
    File targetFile = null;
    try {
        targetFile = moveFile(file, Paths.get(file.getParent(), "..", "backup",
            "success"));
    } catch (IOException e) {
        e.printStackTrace();
    }
}

```

```

    return targetFile;
}

private static File moveToDeadDir(File file) {
    File targetFile = null;
    try {
        targetFile = moveFile(file, Paths.get(file.getParent(), "..", "backup",
            "dead"));
    } catch (IOException e) {
        e.printStackTrace();
    }
    return targetFile;
}

private static File moveFile(File srcFile, Path destPath) throws IOException {
    // 完整代码见本书配套下载资源
}

class UploadTask implements Callable<File> {
    private final File file;

    public UploadTask(File file) {
        this.file = file;
    }

    @Override
    public File call() throws Exception {
        Debug.info("uploading %s", file.getCanonicalPath());
        // 上传指定的文件
        upload(file);
        return file;
    }
}

// 初始化 FTP 客户端
public void init() throws Exception {
    // 完整代码见本书配套下载资源
}

// 将指定的文件上传至 FTP 服务器
protected void upload(File file) throws Exception {
    // 完整代码见本书配套下载资源
}

@Override
public void close() throws IOException {
    // 完整代码见本书配套下载资源
}
}

```

`ExecutorService.invokeAll(Collection<? extends Callable<T>> tasks)`也能够用来批量提交异步任务，该方法能够并发执行 `tasks` 参数所指定的一批任务，但是该方法只有在 `tasks` 参数所指定的一批任务中的所有任务都执行结束之后才返回，其返回值是一个包含各个任务对应的 `Future` 实例的列表（`List`）。因此，使用 `invokeAll` 方法提交批量任务的时候，任务提交方等待 `invokeAll` 方法返回的时间取决于这批任务中最耗时的任务的执行耗时。

9.3 异步计算助手：FutureTask

无论是 `Runnable` 实例还是 `Callable` 实例所表示的任务，只要我们将其提交给线程池执行，那么这些任务就是异步任务。采用 `Runnable` 实例来表示异步任务，其优点是任务既可以交给一个专门的工作者线程执行（以相应的 `Runnable` 实例为参数创建并启动一个工作者线程），也可以交给一个线程池或者 `Executor` 的其他实现类来执行；其缺点是我们无法直接获取任务的执行结果。使用 `Callable` 实例来表示异步任务，其优点是我们可以通过 `ThreadPoolExecutor.submit(Callable<T>)` 的返回值获取任务的执行结果；其缺点是 `Callable` 实例表示的异步任务只能交给线程池执行，而无法直接交给一个专门的工作者线程或者 `Executor` 实现类执行。因此，使用 `Callable` 实例来表示异步任务会使任务执行方式的灵活性大为受限。

`java.util.concurrent.FutureTask` 类则融合了 `Runnable` 接口和 `Callable` 接口的优点：`FutureTask` 是 `Runnable` 接口的一个实现类，因此 `FutureTask` 表示的异步任务可以交给专门的工作者线程执行，也可以交给 `Executor` 实例（比如线程池）执行；`FutureTask` 还能够直接返回其代表的异步任务的执行结果。`ThreadPoolExecutor.submit(Callable<T> task)` 的返回值就是一个 `FutureTask` 实例。`FutureTask` 是 `java.util.concurrent.RunnableFuture` 接口的一个实现类。由于 `RunnableFuture` 接口继承了 `Future` 接口和 `Runnable` 接口，因此 `FutureTask` 既是 `Runnable` 接口的实现类也是 `Future` 接口的实现。`FutureTask` 的一个构造器可以将 `Callable` 实例转换为 `Runnable` 实例，该构造器的声明如下：

```
public FutureTask(Callable<V> callable)
```

该构造器使得我们能够方便地创建一个能够返回处理结果的异步任务。我们可以将任务的执行逻辑封装在一个 `Callable` 实例中，并以该实例为参数创建一个 `FutureTask` 实例。由于 `FutureTask` 类实现了 `Runnable` 接口，因此上述构造器的作用就相当于将 `Callable` 实例转换为 `Runnable` 实例，而 `FutureTask` 实例本身也代表了我们要执行的任务。我们可以用 `FutureTask` 实例（`Runnable` 实例）为参数来创建并启动一个工作者线程以执行相应的任务，也可以将 `FutureTask` 实例交给 `Executor` 执行（通过 `Executor.execute(Runnable task)` 调

用)。FutureTask 类还实现了 Future 接口,这使得我们在调用 Executor.execute(Runnable task) 这样只认 Runnable 接口的方法来执行任务的情况下依然能够获取任务的执行结果:一个工作者线程(可以是线程池中的一个工作者线程)负责调用 FutureTask.run() 执行相应的任务,另外一个线程则调用 FutureTask.get() 来获取任务的执行结果。因此,FutureTask 实例可被看作一个异步任务,它使得任务的执行和对任务执行结果的处理得以并发执行,从而有利于提高系统的并发性。

ThreadPoolExecutor.submit(Callable<T> task) 方法继承自 AbstractExecutorService.submit(Callable<T> task)。AbstractExecutorService.submit(Callable<T> task) 内部实现就是借助 FutureTask 的,如图 9-2 所示。submit 方法会根据指定的 Callable 实例 task 创建一个 FutureTask 实例 ftask,并通过 Executor.execute(Runnable) 调用异步执行 ftask 所代表的任务,然后返回 ftask,以便该方法的调用方能够获取任务的执行结果。

```
public <T> Future<T> submit(Callable<T> task) {
    if (task == null) throw new NullPointerException();
    RunnableFuture<T> ftask = newTaskFor(task);
    execute(ftask);
    return ftask;
}
```

图 9-2 AbstractExecutorService.submit(Callable<T>) 源码

FutureTask 还支持以回调 (Callback) 的方式处理任务的执行结果。当 FutureTask 实例所代表的任务执行结束后,FutureTask.done() 会被执行⁵。FutureTask.done() 是个 protected 方法,FutureTask 子类可以覆盖该方法并在其中实现对任务执行结果的处理。FutureTask.done() 中的代码可以通过 FutureTask.get() 调用来获取任务的执行结果,此时由于任务已经执行结束,因此 FutureTask.get() 调用并不会使得当前线程暂停。但是,由于任务的执行结束既包括正常终止,也包括异常终止以及任务被取消而导致的终止,因此 FutureTask.done() 方法中的代码可能需要在调用 FutureTask.get() 前调用 FutureTask.isCancelled() 来判断任务是否被取消,以免 FutureTask.get() 调用抛出 CancellationException 异常(运行时异常),如清单 9-3 所示。

9.3.1 实践:实现 XML 文档的异步解析

Java 标准库所提供的 XML 文档解析器 javax.xml.parsers.DocumentBuilder 仅支持以同

⁵ FutureTask.done() 的执行线程与 FutureTask.run() 的执行线程是同一个线程。

步的方式去解析 XML 文档，这意味着直接使用 `DocumentBuilder` 解析 XML 文档，我们必须等待 XML 文档解析完毕才能从 XML 文档中查询数据。利用 `FutureTask` 我们可以自行实现一个支持异步解析的 XML 解析器 `XMLDocumentParser`，如清单 9-3 所示。

清单 9-3 基于 `FutureTask` 的 XML 异步解析器

```
/**
 * 支持异步解析器的 XML 解析器
 *
 * @author Viscent Huang
 */
public class XMLDocumentParser {

    public static ParsingTask newTask(InputStream in) {
        return new ParsingTask(in);
    }

    public static ParsingTask newTask(URL url) throws IOException {
        return newTask(url, 30000, 30000);
    }

    // 完整代码见本书配套下载资源

    // 封装对 XML 解析结果进行处理的回调方法
    public abstract static class ResultHandler {
        abstract void onSuccess(Document document);

        void onError(Throwable e) {
            e.printStackTrace();
        }
    }

    public static class ParsingTask {
        private final InputStream in;
        private volatile Executor executor;
        private volatile ResultHandler resultHandler;

        public ParsingTask(InputStream in, Executor executor, ResultHandler
            resultHandler) {
            this.in = in;
            this.executor = executor;
            this.resultHandler = resultHandler;
        }

        public ParsingTask(InputStream in) {
            this(in, null, null);
        }
    }
}
```

```

public Future<Document> execute() throws Exception {
    FutureTask<Document> ft;
    final Callable<Document> task = new Callable<Document>() {
        @Override
        public Document call() throws Exception {
            return doParse(in);
        }
    };
    final Executor theExecutor = executor;
    // 解析模式: 异步/同步
    final boolean isAsyncParsing = (null != theExecutor);
    final ResultHandler rh;
    if (isAsyncParsing && null != (rh = resultHandler)) {
        ft = new FutureTask<Document>(task) {
            @Override
            protected void done() {
                // 回调 ResultHandler 的相关方法对 XML 解析结果进行处理
                callbackResultHandler(this, rh);
            }
        }; // FutureTask 匿名类结束
    } else {
        ft = new FutureTask<Document>(task);
    }
    if (isAsyncParsing) {
        theExecutor.execute(ft); // 交给 Executor 执行, 以支持异步执行
    } else {
        ft.run(); // 直接(同步)执行
    }
    return ft;
}

void callbackResultHandler(FutureTask<Document> ft, ResultHandler rh) {
    // 获取任务处理结果前判断任务是否被取消
    if (ft.isCancelled()) {
        Debug.info("parsing cancelled.%s", ParsingTask.this);
        return;
    }
    try {
        Document doc = ft.get();
        rh.onSuccess(doc);
    } catch (InterruptedException ignored) {
        Debug.info("retrieving result cancelled.%s", ParsingTask.this);
    } catch (ExecutionException e) {
        rh.onError(e.getCause());
    }
}

```



```
XMLDocumentParser.newTask(url).setExecutor(executor).setResultHandler(
    new ResultHandler() {
        @Override
        public void onSuccess(Document document) {
            process(document);
        }
    })
    .execute();
```

这里我们指定了一个 `ResultHandler` 以回调的方式来处理 XML 解析结果。同样是异步解析,我们也可以不指定 `ResultHandler`,而是在程序需要 XML 解析结果的时候自己通过 `Future.get()` 调用来获取:

```
Future<Document> future = XMLDocumentParser.newTask(url).setExecutor(es).execute();
doSomething(); // 执行其他操作
process(future.get());
```

如果要采用同步方式解析 XML 解析,我们只需要:

```
Future<Document> future;
future = XMLDocumentParser.newTask(url).execute();
process(future.get()); // 直接获取解析结果进行处理
```

由此可见,在不使用 `ResultHandler` 的情况下,异步解析方式和同步解析方式的客户端代码编写方式几乎是一样的:异步解析方式比同步方式多了一个 `ParsingTask.setExecutor` 方法调用;在异步解析方式下,客户端代码在 `ParsingTask.execute()` 调用与 `Future.get()` 调用之间往往会执行其他操作,以减少因 XML 异步解析未完成而导致 `Future.get()` 调用造成等待的可能性。

从上述分析可知, `FutureTask` 的使用既可以发挥异步编程的好处,又可以在一定程度上屏蔽同步编程与异步编程之间的差异,这简化了代码。

9.3.2 可重复执行的异步任务

`FutureTask` 基本上是被设计用来表示一次性执行的任务,其内部会维护一个表示任务运行状态(包括未开始运行、已经运行结束等)的状态变量, `FutureTask.run()` 在执行任务处理逻辑前会先判断相应任务的运行状态,如果该任务已经被执行过,那么 `FutureTask.run()` 会直接返回(并不会抛出异常)。因此, `FutureTask` 实例所代表的任务是无法被重复执行的。这意味着同一个 `FutureTask` 实例不能多次提交给 `Executor` 实例执行(尽管这样做不会导致异常的抛出)。 `FutureTask.runAndReset()` 能够打破这种限制,使得一个 `FutureTask` 实例所代表的任务能够多次被执行。 `FutureTask.runAndReset()` 是一个 `protected` 方法,它能够

执行 `FutureTask` 实例所代表的任务但是不记录任务的执行结果。因此，如果同一个对象所代表的任务需要被多次执行，并且我们需要对该任务每次的执行结果进行处理，那么 `FutureTask` 仍然是不适用的，此时我们可以考虑使用如清单 9-4 所示的抽象异步任务类 `AsyncTask` 来表示这种任务。

`AsyncTask` 抽象类同时实现了 `Runnable` 接口和 `Callable` 接口。`AsyncTask` 子类通过覆盖 `call` 方法来实现其任务处理逻辑，而 `AsyncTask.run()` 则充当任务处理逻辑的执行入口。`AsyncTask` 实例可以提交给 `Executor` 实例执行。当任务执行成功结束后，相应 `AsyncTask` 实例的 `onResult` 方法会被调用以处理任务的执行结果；当任务执行过程中抛出异常时，相应 `AsyncTask` 实例的 `onError` 方法会被调用以处理这个异常。`AsyncTask` 的子类可以覆盖 `onResult` 方法、`onError` 方法来对任务执行结果、任务执行过程中抛出的异常进行处理。由于 `AsyncTask` 在回调 `onResult`、`onError` 方法的时候不是直接调用而是通过向 `Executor` 实例 `executor` 提交一个任务进行的，因此 `AsyncTask` 的任务执行（即 `AsyncTask.run()` 调用）可以是在一个工作者线程中进行的，而对任务执行结果的处理则可以在另外一个线程中进行，这就从整体上实现了任务的执行与对任务执行结果的处理的并发：设 `asyncTask` 为一个任意 `AsyncTask` 实例，当一个线程在执行 `asyncTask.onResult` 方法处理 `asyncTask` 一次执行的执行结果时，另外一个工作者线程可能正在执行 `asyncTask.run()`，即进行 `asyncTask` 的下一轮执行。

注意

`FutureTask` 所代表的任务无法被多次执行，除非相应的任务是通过调用 `FutureTask.runAndReset()` 方法执行的。

清单 9-4 支持重复执行的异步任务抽象类

```
/**
 * 能够被重复执行的抽象异步任务
 *
 * @author Viscent Huang
 */
public abstract class AsyncTask<V> implements Runnable,
    Callable<V> {
    protected final Executor executor;

    public AsyncTask(Executor executor) {
        this.executor = executor;
    }

    public AsyncTask() {
        this(new Executor() {
            @Override
            public void execute(Runnable command) {
```

```

        command.run();
    }
});
}

@Override
public void run() {
    Exception exp = null;
    V r = null;
    try {
        r = call();
    } catch (Exception e) {
        exp = e;
    }

    final V result = r;
    if (null == exp) {
        executor.execute(new Runnable() {
            @Override
            public void run() {
                onResult(result);
            }
        });
    } else {
        final Exception exceptionCaught = exp;
        executor.execute(new Runnable() {
            @Override
            public void run() {
                onError(exceptionCaught);
            }
        });
    }
} // run 结束

/**
 * 留给子类实现任务执行结果的处理逻辑
 */
* @param result
*     任务执行结果
*/
protected abstract void onResult(V result);

/**
 * 子类可覆盖该方法来对任务执行过程中抛出的异常进行处理
 */
* @param e
*     任务执行过程中抛出的异常
*/

```

```
protected void onError(Exception e) {
    e.printStackTrace();
}
}
```

清单 9-6 展示了 `AsyncTask` 的使用场景。

9.4 计划任务

在有些情况下，我们可能需要事先提交一个任务，这个任务并不是立即被执行的，而是要在指定的时间或者周期性地被执行，这种任务就被称为计划任务（`Scheduled Task`）。典型的计划任务包括清理系统垃圾数据、系统监控、数据备份等。

`ExecutorService` 接口的子类 `ScheduledExecutorService` 接口定义了一组方法用于执行计划任务。`ScheduledExecutorService` 接口的默认实现类是 `java.util.concurrent.ScheduledThreadPoolExecutor` 类，它是 `ThreadPoolExecutor` 的一个子类。`Executors` 除了提供创建 `ExecutorService` 实例的便捷工厂方法之外，它还提供了两个静态工厂方法用于创建 `ScheduledExecutorService` 实例：

```
public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)
public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize,
    ThreadFactory threadFactory)
```

`ScheduledExecutorService` 接口定义的方法按其功能可分为以下两种。

- 延迟执行提交的任务。这包括以下两个方法：

```
<V> ScheduledFuture<V> schedule(Callable<V> callable,long delay,TimeUnit unit)
ScheduledFuture<?> schedule(Runnable command,long delay,TimeUnit unit)
```

上述两个方法使得我们可以采用 `Callable` 实例或者 `Runnable` 实例来表示任务。`delay` 参数和 `unit` 参数一起用来表示被提交的任务自其提交的那一刻到其开始执行之间的时间差，即延时。上述方法的返回值类型 `ScheduledFuture` 继承自 `Future` 接口，因此我们也可以利用上述方法的返回值来获取所提交的计划任务的处理结果。

- 周期性地执行提交的任务。这包括以下两个方法：

```
ScheduledFuture<?> scheduleAtFixedRate(Runnable command,long initialDelay,
    long period,TimeUnit unit)
```

约定

同一个任务任意两次执行的开始时间之间的时间差被称为该任务的执行周期，记为 Interval。

一个任务从其开始执行到其执行结束所需的时间被称为该任务的耗时，简称耗时，记为 Execution Time。

从该方法的名字上看，它能够以固定的频率不断地执行 command 参数所指定的任务。initialDelay 参数和 unit 参数一起指定了一个时间偏移，任务首次执行的开始时间就是任务提交时间加上这个偏移。实际上，提交给 scheduleAtFixedRate 方法执行的计划任务，其执行周期并不一定是固定的，它会同时受 Execution Time 和 period 的影响—— $\text{Interval} = \max(\text{Execution Time}, \text{period})$ ，如图 9-3 所示：如果任务的每次执行总是能够在 period 指定的时间跨度内完成时，那么该任务的执行周期就是 period 指定的时间跨度，此时任务的执行周期是恒定的；如果该任务的某些次执行，其执行耗时超过了 period 指定的时间跨度，那么该任务的执行周期就会变得不固定——有时其执行周期等于 period，有时却大于 period。

```

start-----[-----]____[-----]____[-----]____[...
|initial delay|execution time|idle| execution time || execution time |
    <-      period      -><-      period      -><-      period | extra ->
    <-      interval     -><-      interval     -><-      interval     ->
  
```

图 9-3 scheduleAtFixedRate 方法执行任务的周期示意图

scheduleWithFixedDelay 方法则能够以一定的时间间隔不断地执行 command 所指定的任务。

```

ScheduledFuture<?> scheduleWithFixedDelay(Runnable command, long initialDelay,
                                           long delay, TimeUnit unit)
  
```

其中，initialDelay 参数和 unit 参数一起指定了一个时间偏移，任务首次执行的开始时间就是任务提交时间加上这个偏移。提交给 scheduleWithFixedDelay 方法执行的计划任务的执行周期 $\text{Interval} = \text{Execution Time} + \text{delay}$ ，其中 delay 是一个固定值，因此任务的执行周期实际上也不是固定的而是随 Execution Time 的变化而变化，如图 9-4 所示。

```

start-----[-----]____[-----]____[-----]____[...
|initial delay|execution time|delay| execution time |delay|| execution time |delay|
    <-      interval     -><-      interval     -><-      interval     ->
  
```

图 9-4 scheduleWithFixedDelay 方法执行任务的周期示意图

由于同一个任务每次执行的耗时可能都不同，它既可能变大也可能变小。因此，

Execution Time 值有可能比 period 或者 delay 的参数值还大。这就导致了同一个任务的执行周期往往不是固定的，如清单 9-5 所示的 Demo 能够展示这一点。

清单 9-5 ScheduledExecutorService 使用 Demo

```
public class ScheduledTaskDemo {
    static ScheduledExecutorService ses = Executors.newScheduledThreadPool(2);

    public static void main(String[] args) throws InterruptedException {
        final int argc = args.length;
        // 任务执行最大耗时
        int maxConsumption;
        // 任务执行最小耗时
        int minConsumption;
        if (argc >= 2) {
            minConsumption = Integer.valueOf(args[0]);
            maxConsumption = Integer.valueOf(args[1]);
        } else {
            maxConsumption = minConsumption = 1000;
        }
        ses.scheduleAtFixedRate(new SimulatedTask(minConsumption, maxConsumption,
            "scheduleAtFixedRate"), 0, 2, TimeUnit.SECONDS);
        ses.scheduleWithFixedDelay(new SimulatedTask(minConsumption,
            maxConsumption,
            "scheduleWithFixedDelay"), 0, 1, TimeUnit.SECONDS);
        Thread.sleep(20000);

        ses.shutdown();
    }

    static class SimulatedTask implements Runnable {
        private String name;
        // 模拟任务执行耗时
        private final int maxConsumption;
        private final int minConsumption;
        private final AtomicInteger seq = new AtomicInteger(0);

        public SimulatedTask(int minConsumption, int maxConsumption, String name) {
            this.maxConsumption = maxConsumption;
            this.minConsumption = minConsumption;
            this.name = name;
        }

        @Override
        public void run() {
            try {
                // 模拟任务执行耗时
            }
        }
    }
}
```

```

    Tools.randomPause(maxConsumption, minConsumption);
    Debug.info(name + " run-" + seq.incrementAndGet());
} catch (Exception e) {
    e.printStackTrace();
}
}
} // run 结束
}
}

```

在 Execution Time 始终不长于 delay 或者 period 所代表的时间的情况下, scheduleAtFixedRate 和 scheduleWithFixedDelay 能够实现同样的效果——按照固定的时间间隔不断地执行任务。例如, 使用如下命令运行清单 9-5 所示的程序:

```
java io.github.viscent.mtia.ch9.ScheduledTaskDemo 1000 1000
```

上述命令的输出类似如下:

```

[2016-06-23 20:29:33.959] [INFO] [pool-1-thread-2]:scheduleWithFixedDelay run-1
[2016-06-23 20:29:33.959] [INFO] [pool-1-thread-1]:scheduleAtFixedRate run-1
[2016-06-23 20:29:35.878] [INFO] [pool-1-thread-1]:scheduleAtFixedRate run-2
[2016-06-23 20:29:35.962] [INFO] [pool-1-thread-2]:scheduleWithFixedDelay run-2
[2016-06-23 20:29:37.878] [INFO] [pool-1-thread-1]:scheduleAtFixedRate run-3
[2016-06-23 20:29:37.962] [INFO] [pool-1-thread-2]:scheduleWithFixedDelay run-3
[2016-06-23 20:29:39.878] [INFO] [pool-1-thread-1]:scheduleAtFixedRate run-4
[2016-06-23 20:29:39.963] [INFO] [pool-1-thread-2]:scheduleWithFixedDelay run-4

```

可见, 两个计划任务都是每 2 秒执行一次。

在 Execution Time 长于 delay 或者 period 所代表的时间的情况下, scheduleAtFixedRate 和 scheduleWithFixedDelay 都无法保证计划任务以固定的周期被执行。例如, 使用如下命令运行如清单 9-5 所示的程序:

```
java io.github.viscent.mtia.ch9.ScheduledTaskDemo 1000 3000
```

上述命令的输出类似如下:

```

[2016-06-23 20:06:19.417] [INFO] [pool-1-thread-1]:scheduleWithFixedDelay run-1
[2016-06-23 20:06:19.650] [INFO] [pool-1-thread-2]:scheduleAtFixedRate run-1
[2016-06-23 20:06:21.227] [INFO] [pool-1-thread-2]:scheduleAtFixedRate run-2
[2016-06-23 20:06:22.774] [INFO] [pool-1-thread-1]:scheduleWithFixedDelay run-2
[2016-06-23 20:06:23.832] [INFO] [pool-1-thread-2]:scheduleAtFixedRate run-3
[2016-06-23 20:06:25.378] [INFO] [pool-1-thread-1]:scheduleWithFixedDelay run-3
[2016-06-23 20:06:26.097] [INFO] [pool-1-thread-2]:scheduleAtFixedRate run-4
[2016-06-23 20:06:27.214] [INFO] [pool-1-thread-2]:scheduleAtFixedRate run-5
[2016-06-23 20:06:28.863] [INFO] [pool-1-thread-1]:scheduleWithFixedDelay run-4

```

可见，两个任务的执行周期分别在 1~3 和 2~4 之间变化。从以上输出中还可以看出，一个任务的执行耗时超过 `period` 或者 `delay` 所表示的时间只会导致该任务的下一次执行时间被相应地推迟，而不会导致该任务在同一个时间内被运行多次（并发执行）。

注意 一个任务的执行耗时超过 `period` 或者 `delay` 所表示的时间只会导致该任务的下一次执行时间被相应地推迟，而不会导致该任务被并发执行。

任务执行结果处理、异常处理与任务取消

延迟执行的任务最多只会被执行一次，因此我们利用 `schedule` 方法的返回值（`ScheduledFuture` 实例）便能获取这种计划任务的执行结果、执行过程中抛出的异常以及取消任务的执行。

周期性执行的任务会不断地被执行，直到任务被取消或者相应的 `ScheduledExecutorService` 实例被关闭。因此，`scheduleAtFixedRate` 方法、`scheduleWithFixedDelay` 方法的返回值（`ScheduledFuture`）能够取消相应的任务，但是它无法获取计划任务的一次或者多次的执行结果⁶。如果我们需要对周期性执行的计划任务的执行结果进行处理，那么可以考虑使用如清单 9-4 所示的异步任务类 `AsyncTask` 来表示计划任务。

清单 9-6 使用 `AsyncTask` 模拟了这样一个周期性任务：每隔一段时间（比如 3 秒）检测一下当前主机与指定的目标主机之间的网络连通性（比如使用 `ping` 命令检测），并将检测的结果记录到数据库之中。这里，我们在 `AsyncTask` 的匿名子类的 `call` 方法中实现检测的逻辑，并在 `onResult` 方法中将检测的结果记录到数据库之中。可见，这个计划任务的任务执行逻辑和结果处理逻辑是异步进行的。

清单 9-6 周期性任务的执行结果处理 Demo

```
public class PeriodicTaskResultHandlingDemo {
    final static ScheduledExecutorService ses = Executors.newScheduledThreadPool(2);

    public static void main(String[] args) throws InterruptedException {
        final String host = args[0];
        final AsyncTask<Integer> asyncTask = new AsyncTask<Integer>(ses) {
            final Random rnd = new Random();
```

6 周期性执行的任务会不断地被执行，因此获取这种计划任务任意一次执行的结果意义不大，而获取全部次执行的结果又有些困难——只有当相应的计划任务不会再被执行的情况下我们才能够获取这样的结果。

```

final String targetHost = host;

@Override
public Integer call() throws Exception {
    return pingHost();
}

private Integer pingHost() throws Exception {
    // 模拟实际操作耗时
    Tools.randomPause(2000);
    // 模拟的探测结果码
    Integer r = Integer.valueOf(rnd.nextInt(4));
    return r;
}

@Override
protected void onResult(Integer result) {
    // 将结果保存到数据库
    saveToDatabase(result);
}

private void saveToDatabase(Integer result) {
    Debug.info(targetHost + " status:" + String.valueOf(result));
    // 省略其他代码
}

@Override
public String toString() {
    return "Ping " + targetHost + "," + super.toString();
}
};

ses.scheduleAtFixedRate(asyncTask, 0, 3, TimeUnit.SECONDS);
}
}

```

提交给 `ScheduledExecutorService` 执行的计划任务在其执行过程中如果抛出未捕获的异常 (`Uncaught Exception`), 那么该任务后续就不会再被执行。即使我们在创建 `ScheduledExecutorService` 实例的时候指定一个线程工厂, 并使线程工厂为其创建的线程关联一个 `UncaughtExceptionHandler`, 当计划任务抛出未捕获异常的时候该 `UncaughtExceptionHandler` 也不会被 `ScheduledExecutorService` 实例调用。因此, 我们必须确保周期性执行的任务在其执行过程中不会抛出任何未捕获异常。

注意

提交给 `ScheduledExecutorService` 执行的计划任务在其执行过程中如果抛出未捕获的异常 (`Uncaught Exception`), 那么该任务后续就不会再被执行。

9.5 本章小结

本章介绍了同步计算与异步计算的概念，并介绍了 Java 平台对异步计算所提供的相关 API。本章知识结构如图 9-5 所示。

从单个任务的角度来看，任务的执行方式可以是同步的，也可以是异步的。同步方式的优点是代码简单、直观，缺点是它往往意味着阻塞，因此不利于系统的吞吐率。异步方式的优点则是它往往意味着非阻塞，因此有利于系统的吞吐率，其代价是相对复杂的代码和额外的开销。阻塞/非阻塞是任务执行方式的属性，它们与任务执行方式没有必然的联系：同步任务既可能是阻塞的，也可能是非阻塞的；异步任务既可能是非阻塞的，也可能是阻塞的。对于同一个任务，我们既可以说它是同步任务也可以说它是异步任务，这取决于任务的执行方式以及我们的观察角度。

Runnable/Callable 接口是对任务处理逻辑进行的抽象，而 Executor 接口是对任务的执行进行的抽象。Executor 接口使得我们能够对任务的提交与任务的具体执行细节进行解耦，这为更改任务的具体执行细节提供了灵活性与便利。ExecutorService 接口是对 Executor 接口的增强：它支持返回异步任务的处理结果、支持资源的管理接口、支持批量任务提交等。ThreadPoolExecutor 是 Executor/ExecutorService 接口的一个实现类。实用工具类 Executors 为线程池的创建提供了快捷方法。CompletionService 接口为异步任务的批量提交以及获取这些任务的处理结果提供了便利，其默认实现类为 ExecutorCompletionService。

FutureTask 是 Java 标准库提供的 Future 接口实现类，它还实现了 Runnable 接口。因此，FutureTask 可直接用来获取异步任务的处理结果，它可以交给专门的工作者线程执行，也可以交给 Executor 实例执行，甚至由当前线程直接执行（同步）。一般来说，FutureTask 是一次性使用的，一个 FutureTask 实例代表的任务只能够被执行一次。如果需要多次执行同一个任务，那么可以考虑本书介绍的 AsyncTask 类。

计划任务的执行方式包括延迟执行和周期性执行。ScheduledThreadPoolExecutor 是 ScheduledExecutorService 接口的默认实现类，它可以用于执行计划任务。ScheduledFuture 接口可用来获取延迟执行的计划任务的处理结果。如果要获取周期性执行的计划任务的处理结果，可以使用自定义的 AsyncTask 类。周期性执行的计划任务，其执行周期并不是固定的，而是受任务单次执行耗时的影响：提交给 scheduleAtFixedRate 方法执行的计划任务，其执行周期为 $\max(\text{Execution Time}, \text{period})$ ；提交给 scheduleWithFixedDelay 方法执行的计划任务，其执行周期为 $\text{Execution Time} + \text{delay}$ 。计划任务在其执行过程中如果抛出未捕获的异常，那么该任务将不会再被执行。

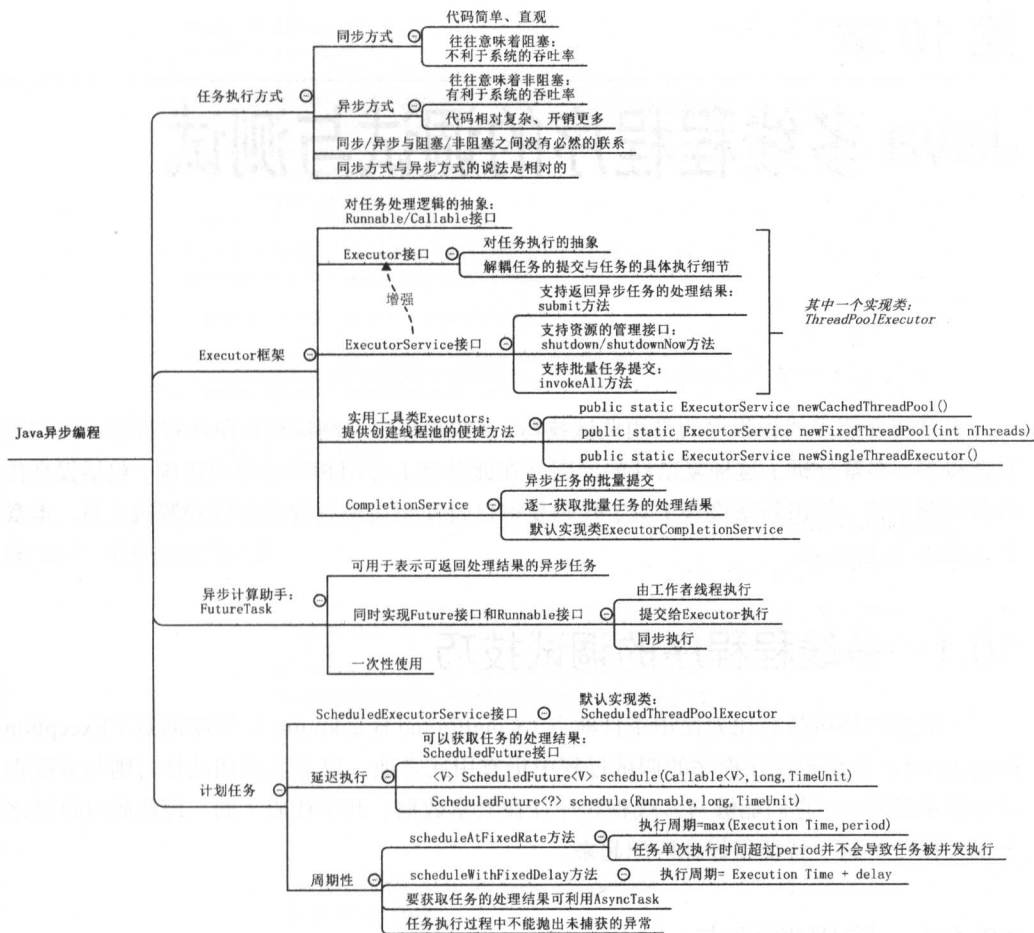


图 9-5 本章知识结构图

第 10 章

Java 多线程程序的调试与测试

本章将介绍多线程程序的常用调试技巧。多线程程序的测试相比单线程程序的测试要复杂得多，本章分析了这种复杂性的原因并在此基础上介绍相应的应对措施，包括提高代码的可测试性、使用静态检查工具、代码复审以及使用简单有效的多线程测试工具。本章默认 IDE 为 Eclipse。

10.1 多线程程序的调试技巧

一般性的调试技巧比如使用条件断点（Conditional Breakpoint）、异常断点（Exception Breakpoint）等在多线程程序的调试过程中仍有用武之地。以下几点调试技巧则与多线程程序联系紧密——它们能够与多线程程序往往共享数据、共享代码（同一段代码可以被多个线程执行）这一特征很好地匹配起来。

10.1.1 使用监视点

多线程程序往往需要共享数据，而这正是线程安全问题产生的前提。因此，在调试过程中理清程序对共享变量（实例变量、静态变量）的访问情况，尤其是更新的情况对问题定位十分有益。Eclipse 的调试功能支持一种名为监视点（Watch Point）的特殊断点，监视点使得我们可以方便地通过调用栈（Call Stack）观察到程序对共享变量的读取、更新情况。例如，在清单 5-6 中的 AbstractService 类的 started 实例变量声明语句上设置一个监视点（断点）会使访问该变量的线程被暂挂，如图 10-1 所示。

有时候我们更加关心线程对共享变量的更新情况，此时为了提高调试效率我们可以将监视点设置为只针对更新操作，如图 10-2 所示。

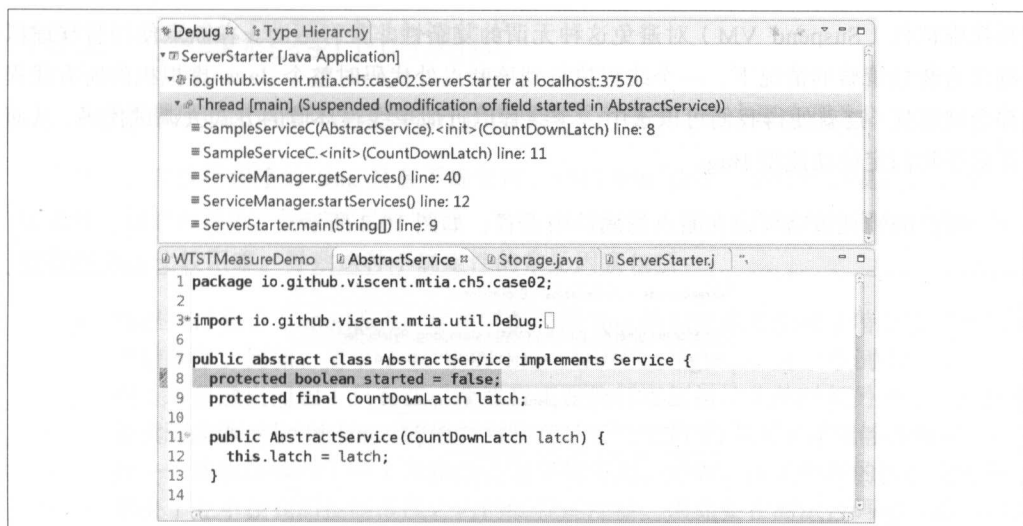


图 10-1 监视点使用示例

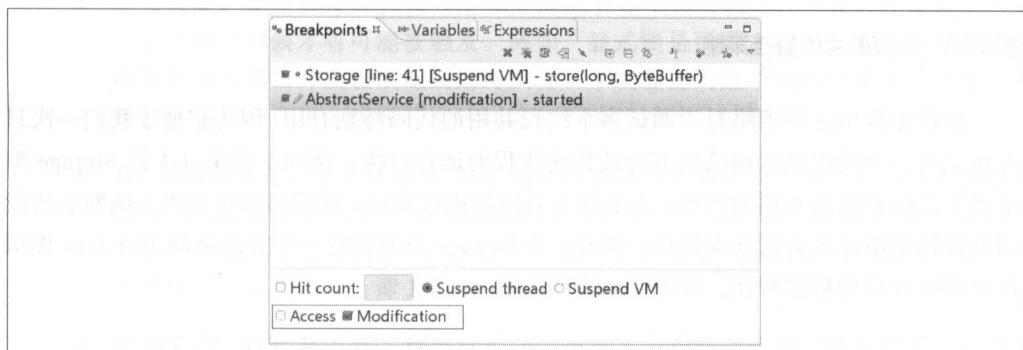


图 10-2 只跟踪更新操作的监视点

10.1.2 设置暂挂策略

Eclipse 断点默认使用的暂挂策略（Suspend Policy）是暂挂线程（Suspend Thread），该暂挂策略只会暂挂执行到断点处代码的线程，而其他线程则仍然可以继续运行。由于多线程程序尤其是新开发的多线程程序本身可能仍然包含一些功能型 Bug——与多线程无关的 Bug，因此在调试过程中我们首先要找出并修复这类 Bug，在此基础上如果程序的行为仍然没有符合我们的期望，那么我们便很容易断定是线程安全问题导致的。暂挂线程这种暂挂策略使得我们在调试一个线程的时候，该线程所访问的共享变量仍然可以被其他线程更新，从而增加了调试的复杂性——我们需要考虑线程安全问题。另外一种暂挂策略——

暂挂虚拟机（Suspend VM）对避免这种无谓的复杂性非常有帮助。在断点使用暂挂虚拟机作为暂挂策略的情况下，一个线程执行到该断点处代码时整个 Java 虚拟机的所有线程都会被暂挂，这就使得我们可以选中一个线程以近似单线程环境的方式去调试代码，从而有利于我们定位功能型 Bug。

断点的暂挂策略可以在断点的属性中设置，如图 10-3 所示。

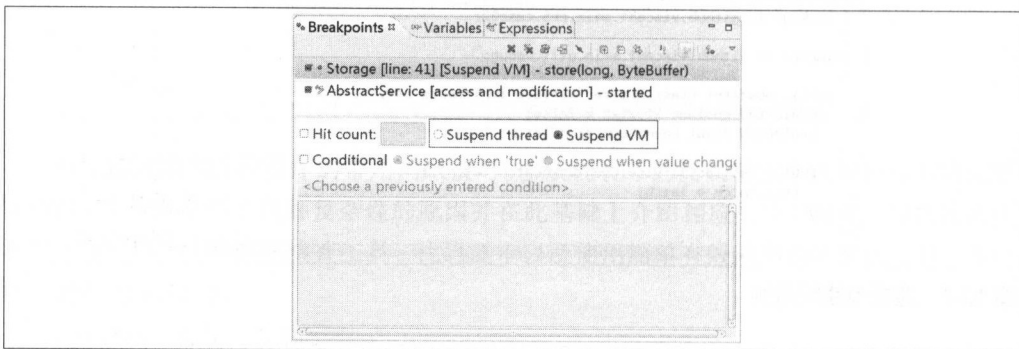


图 10-3 设置断点的暂挂策略

暂挂虚拟机这种策略对于调试多个线程共用的代码特别有用，因为它便于我们一次只专注于对一个线程进行调试而不会被其他线程的运行打扰。例如，清单 4-4 的 Storage 类是多个工作者线程共用的代码，为了便于对该类进行调试，我们可以将该类上的断点所使用的暂挂策略设置为暂挂虚拟机。因此，当 Storage 类的任意一个方法被调用时 Java 虚拟机中的所有线程都被暂挂，如图 10-4 所示。

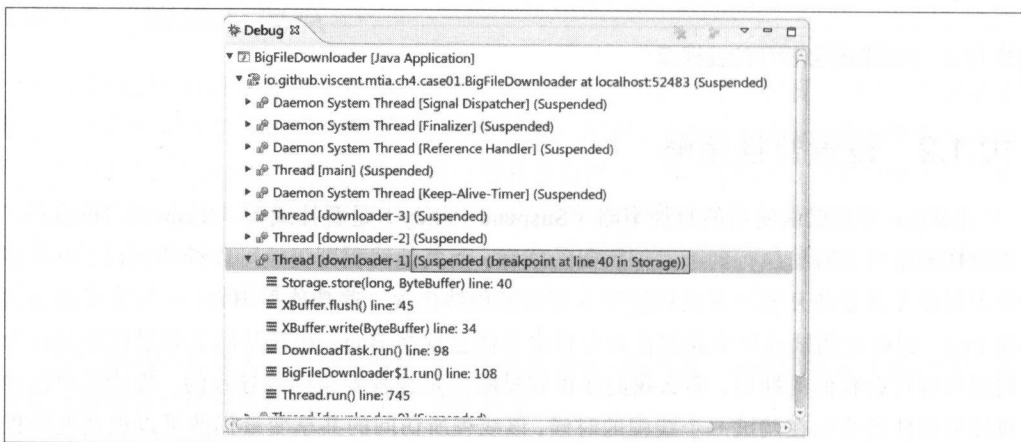


图 10-4 暂挂虚拟机策略的调试效果

10.2 多线程程序的测试

多线程程序的测试相比单线程程序的测试要复杂得多，这主要表现为以下几个方面。

第一，多线程程序的 Bug 具有不确定性，这些 Bug 往往不是必然出现的，而是在一定条件下偶然出现的。例如可见性问题、死锁、饥饿等问题都不是必然出现的。再现多线程程序 Bug 比较困难。导致这种不确定性的常见因素包括：

- 线程执行交错（Interleave）顺序。多线程 Bug 的触发往往依赖于特定的线程执行交错顺序。例如，清单 3-16 中的 `IncorrectDCLSingleton` 这个单例类中存在的问题（对象逸出问题）只有在一个线程正在执行实例初始化的时候另外一个线程恰好读取实例变量 `instance` 的值并判断其是否为 `null` 的情况下才能够被触发——另外一个线程读取到了一个未初始化完毕的实例。然而，由于线程的执行交错顺序取决于并不在我们控制范围之内的线程调度器，因此要在测试代码中构造一定的线程执行交错顺序是很困难甚至是不可能的。
- 并发程度。有些 Bug 在并发程度低的情况下并不会被触发，只有当并发程度达到一定级别的时候才有可能被触发。例如，显式锁是可再入的这一特性使得显式锁的锁泄漏问题在并发程度低到始终只有一个线程访问这个锁的时候基本上无法被触发，而只有在多个线程并发访问这个锁的时候才会被触发¹。
- JIT 编译器优化。某些多线程 Bug 可能需要在 JIT 编译器优化介入之后才能够被触发。例如，清单 2-7 的 `VisibilityDemo` 类中的共享变量 `toCancel` 没有采用 `volatile` 修饰可能导致可见性问题。但是，该问题只有在对 `toCancel` 进行判断的 `while` 循环被执行 1 万次以上，即 JIT 编译器优化介入之后才能够被触发²。
- 硬件平台。有些 Bug 在一种硬件平台上很难甚至无法被触发，而在另外一些硬件平台上则可能很容易被触发。例如，由于 x86 架构的处理器仅支持 `StoreLoad`（写后读）这种内存重排序，因此内存重排序问题往往很难甚至无法在基于 x86 架构的宿主机平台上被触发。这往往造成一种程序不具有这方面 Bug 的假象。

第二，多线程程序同时受线程安全问题（非功能性问题）以及功能性问题的影响。多线程程序在测试阶段可能仍然包含的功能性问题也同样会影响测试，这也增加了测试的难度。

1 同一个线程在持有一个显式锁的情况下最多可以重复 2 147 483 647 次申请该锁，超过这个次数会导致 `Error` 被抛出。

2 这里依照 Java 虚拟机的默认配置而言。这个阈值可以通过 Java 虚拟机参数 “-XX:CompileThreshold” 进行修改。

第三，缺乏成熟的测试工具。针对多线程程序的测试框架、工具之中目前还没有像 JUnit 这样广为接受的，然而 JUnit 本身并没有提供对多线程程序进行测试的支持。

应对多线程程序测试的困难性的常用措施包括：提高代码的可测试性、使用静态检查工具、代码复审以及选用简单有效的多线程测试工具。

10.2.1 可测试性

- 提高多线程程序的可测试性（Testability）可以从以下几个方面入手。
- 抽象（Abstraction）与实现（Implementation）分离。抽象与实现分离是面向对象编程的基本原则——面向接口编程，它不仅可以提高代码的可读性和可扩展性，同时也能提高代码的可测试性。例如，第 4 章的第 2 个实战案例（响应延时统计）充分体现了这一点，该案例中使用的抽象与实现如表 10-1 所示。

表 10-1 响应延时统计程序中抽象与实现分离

抽象（接口/抽象类）	含 义	对应的实现类	好 处
AbstractStatTask	对统计程序的算法步骤进行抽象。该抽象使得我们从单线程版程序“进化”到多线程版程序的过程中无须修改现有代码，而只需要新增一个实现类（MultithreadedStatTask）	SimpleStatTask（单线程版）、MultithreadedStatTask（多线程版）	提高扩展性、可测试性
StatProcessor	对统计处理逻辑的抽象。该抽象便于我们对 AbstractStatTask 实现类、AbstractLogReader 实现类进行单元测试	RecordProcessor	提高可测试性
AbstractLogReader	对日志读取逻辑进行抽象。该抽象便于我们对 AbstractStatTask 实现类进行单元测试	LogReaderThread	提高可测试性

- 数据与数据来源分离。程序所处理的数据可以来自用户输入、文件、数据库以及网络等，而对数据的处理逻辑代码应该只关心数据本身，而不应该关心数据的来源。这种数据与其来源的分离可被看作抽象与实现分离的一个具体应用，它可以降低耦合性（Coupling），并提高代码的灵活性和可测试性。例如，在第 4 章的第 2 个实战案例（响应延时统计）中，尽管该程序的输入数据来自文件（接口日志文件），但是负责读取日志文件记录的实现类 LogReaderThread（代码参见清单 4-10）本身并不直接使用 File 或者 FileInputStream 而是使用 InputStream 来表示其

输入，如下代码片段所示：

```
public LogReaderThread(InputStream in, int inputBufferSize, int batchSize) {
    super(in, inputBufferSize, batchSize);
}
```

这使得在对 LogReaderThread 进行单元测试的时候，我们可以根本不借助文件而是直接使用一个普通对象来表示输入数据（一组日志记录），如清单 10-1 所示。

清单 10-1 LogReaderThread 单元测试示例 JUnit 代码

```
public class LogReaderThreadTest {
    private LogReaderThread logReader;
    private StringBuilder sdb;

    @Before
    public void setUp() throws Exception {
        sdb = new StringBuilder();
        sdb.append("2016-03-30 09:33:04.644|SOAP|request|SMS|sendSms|OSG|ESB|0020000
0000|192.168.1.102|13612345678|136712345670");
        sdb.append("\n2016-03-30 09:33:04.688|SOAP|response|SMS|sendSmsRsp|ESB|OSG|0
02000000000|192.168.1.102|13612345678|136712345670");
        sdb.append("\n2016-03-30 09:33:04.732|SOAP|request|SMS|sendSms|ESB|NIG|00210
000001|192.168.1.102|13612345678|136712345670");
        sdb.append("\n2016-03-30 09:33:04.772|SOAP|response|SMS|sendSmsRsp|NIG|ESB|0
02100000004|192.168.1.102|13612345678|136712345670\n");

        InputStream in = new ByteArrayInputStream(sdb.toString().getBytes("UTF-8"));
        logReader = new LogReaderThread(in, 1024, 4);
        logReader.start();
    }

    @After
    public void tearDown() throws Exception {
        logReader.interrupt();
    }

    @Test
    public void testNextBatch() {
        try {
            RecordSet rs = logReader.nextBatch();
            StringBuilder contents = new StringBuilder();
            String record;
            while (null != (record = rs.nextRecord())) {
                contents.append(record).append("\n");
            }
            assertTrue(contents.toString().equals(sdb.toString()));
        }
    }
}
```

```

    } catch (InterruptedException ignored) {
    }
}
}

```

- 依赖注入（Dependency Injection）。在抽象与实现分离的基础上我们可以进一步实现依赖注入。所谓依赖注入就是指一个对象关联（通常是通过实例变量）另外一个对象（依赖）的时候，该对象并不直接创建其依赖对象，而是通过第三方向其提供（注入）相应对象而实现的。依赖注入使得我们对一个对象进行单元测试时可以使用一个测试桩（Stub）对象来替代该对象的真实依赖，从而简化了单元测试。例如，在第4章的第2个实战案例（响应延时统计）中，AbstractStatTask类（参见清单4-5）的一个构造器允许我们在创建实例时指定其依赖 StatProcessor 实例，该构造器使得我们在单元测试时可以不使用 StatProcessor 接口的现有实现类 RecordProcessor 而是指定一个测试桩对象。
- 关注点分离（Separation Of Concern）。在多线程程序中，将程序中的功能型关注点（Functional Concern）与线程相关的性能关注点（Performance Concern）分离可以极大地提高代码的可测试性。例如，第4章第2个实战案例（响应延时统计）实现的多线程程序的核心功能是，根据指定的日志记录统计外部系统的响应延时情况。该核心功能由本身完全是依照单线程模型来写的 RecordProcessor 类（代码见本书配套下载资源）实现的，因此 RecordProcessor 类完全可以按照单线程的方式进行单独测试（比如使用 Junit），其测试通过则意味着该程序的核心功能测试通过。而该程序直接与线程打交道的代码只有 MultithreadedStatTask 类（参见清单4-7）和 LogReaderThread 类（参见清单4-10），这就使得对这些类进行单元测试时我们只需要关心这些代码本身所需完成的处理而无须关心该程序的核心功能（它应该落实在 RecordProcessor 类的单元测试上），从而降低了测试难度。清单10-2展示了 MultithreadedStatTask 类的单元测试用例代码。MultithreadedStatTask 的一个构造器允许我们指定一个 StatProcessor 接口（其作用参见表10-1）实现：

```
public MultithreadedStatTask(int sampleInterval, StatProcessor recordProcessor)
```

在此，由于 MultithreadedStatTask 类才是我们的单元测试目标，因此在创建 MultithreadedStatTask 实例 mt 的时候，我们并不指定 StatProcessor 接口的真实实现类 RecordProcessor（参见表10-1），而是使用测试桩类 FakeProcessor 的实例。

MultithreadedStatTask.createLogReader()会创建 MultithreadedStatTask 读取日志记录所需的 AbstractLogReader 实例。此时，由于 AbstractLogReader 实例的真实实现类 LogReaderThread（我们已经使用清单10-1中的测试用例对其进行了单元测试）并非我们的测试目标，因此我们并不直接创建 MultithreadedStatTask 实例，而是创建 MultithreadedStatTask 的一个匿名子类，并在该匿名类中覆盖

MultithreadedStatTask.createLogReader(), 使其返回的 AbstractLogReader 实例为一个测试桩类实例 (AbstractLogReader 类的一个匿名子类)。

这里, 我们为单元测试目标类 MultithreadedStatTask 所依赖的其他对象 (包括 StatProcessor 实例和 AbstractLogReader 实例) 都创建了相应的测试桩 (Stub) 对象, 从而降低了测试难度。相反, 如果该程序的核心功能是直接夹杂在 MultithreadedStatTask 类之中的, 那么我们对 MultithreadedStatTask 类进行单元测试时势必隐含着对核心功能的测试。

清单 10-2 MultithreadedStatTask 单元测试 JUnit 源码

```
public class MultithreadedStatTaskTest {
    private MultithreadedStatTask mst;
    private int recordCount = 0;
    private String[] records;

    @Before
    public void setUp() throws Exception {
        records = new String[4];
        records[0] = "2016-03-30 09:33:04.644|SOAP|request|SMS|sendSms|OSG|ESB|0020000000|192.168.1.102|13612345678|136712345670";
        records[1] = "2016-03-30 09:33:04.688|SOAP|response|SMS|sendSmsRsp|ESB|OSG|0200000000|192.168.1.102|13612345678|136712345670";
        records[2] = "2016-03-30 09:33:04.732|SOAP|request|SMS|sendSms|ESB|NIG|0021000001|192.168.1.102|13612345678|136712345670";
        records[3] = "2016-03-30 09:33:04.772|SOAP|response|SMS|sendSmsRsp|NIG|ESB|0210000004|192.168.1.102|13612345678|136712345670";
        mst = createTask(10, 3, "sendSms", "*");
    }

    @After
    public void tearDown() throws Exception {
        recordCount = 0;
    }

    @Test
    public void testRun() {
        // 只关心 MultithreadedStatTask 本身 (与多线程有关)
        mst.run();
        assertTrue(records.length == recordCount);
    }

    private MultithreadedStatTask createTask(
        int sampleInterval,
        int traceIdDiff, String expectedOperationName,
        String expectedExternalDeviceList) throws Exception {
```

```

// Stub 对象
final AbstractLogReader logReader = new AbstractLogReader(
    new ByteArrayInputStream(new byte[] {}), 1024, 4) {
    boolean eof = false;
    RecordSet consumedBatch = new RecordSet(super.batchSize);

    @Override
    protected RecordSet getNextToFill() {
        return null;
    }

    @Override
    protected RecordSet nextBatch() {
        if (eof) {
            return null;
        }
        for (String r : records) {
            consumedBatch.putRecord(r);
        }
        eof = true;
        return consumedBatch;
    }

    @Override
    protected void publish(RecordSet recordBatch) {
        // 什么也不做
    }

    @Override
    public void run() {
        // 什么也不做
    }
};

// 返回 MultithreadedStatTask 的匿名子类
return new MultithreadedStatTask(sampleInterval, new FakeProcessor()) {
    @Override
    protected AbstractLogReader createLogReader() {
        // 并不返回 AbstractLogReader 类的真实实现类 LogReaderThread，而是一个 Stub 类实例
        return logReader;
    }
}; // 不使用 StatProcessor 的真实实现类 RecordProcessor，而是使用 Stub 类 FakeProcessor
// createTask 结束

// Stub 类
class FakeProcessor implements StatProcessor {
    @Override

```

```

public void process(String record) {
    recordCount++;
}

@Override
public Map<Long, DelayItem> getResult() {
    // 不关心该方法，故返回空的 Map
    return Collections.emptyMap();
}
} // FakeProcessor 结束
}

```

- 使工作者线程数可以配置。多线程 Bug 的触发往往与程序的并发程度有关，因此使程序中的工作者线程数量可以配置，便于我们在测试中动态调整线程数以提高或者降低并发程度。

10.2.2 静态检查工具：FindBugs

FindBugs 是一款基于字节码（Byte Code）检查的开源免费静态检查工具³。它可以帮助我们无须运行代码的情况下（但是，代码必须是可编译的）检查出多线程程序中的一些常见错误⁴，包括 `Object.wait()/Condition.await()` 调用没有放在循环体中，`ReentrantLock.unlock()` 调用没有放在 `finally` 块中，在构造器中启动一个线程，等等。例如，FindBugs 可以检查出清单 3-16 中的 `IncorrectDCLSingleton` 类对双重检查锁定的使用可能是错误的，如图 10-5 所示。

当然工具也有局限性，FindBugs 的检查结果也有误报的时候。例如，清单 3-7 中的 `Counter` 类实际上是线程安全的，尽管 `Counter.increment` 方法中包含了 `volatile` 变量 `count` 的自增操作（`count++`），但是这个操作实际上是在临界区中执行的。然而，FindBugs 却以“`volatile` 变量自增操作不具有原子性”的理由认为这段代码是错误的。对于这种误报，我们可以通过使用 FindBugs 提供的注解 `SuppressWarnings` 对相应的方法进行标记以屏蔽相应的警告⁵，如图 10-6 所示。

3 FindBugs 可以从 <http://findbugs.sourceforge.net/> 下载。

4 详见：<http://findbugs.sourceforge.net/bugDescriptions.html>。

5 也可以使用 FindBugs 的 Filter 机制进行批量忽略误报警告，详见：<http://findbugs.sourceforge.net/manual/filter.html>。

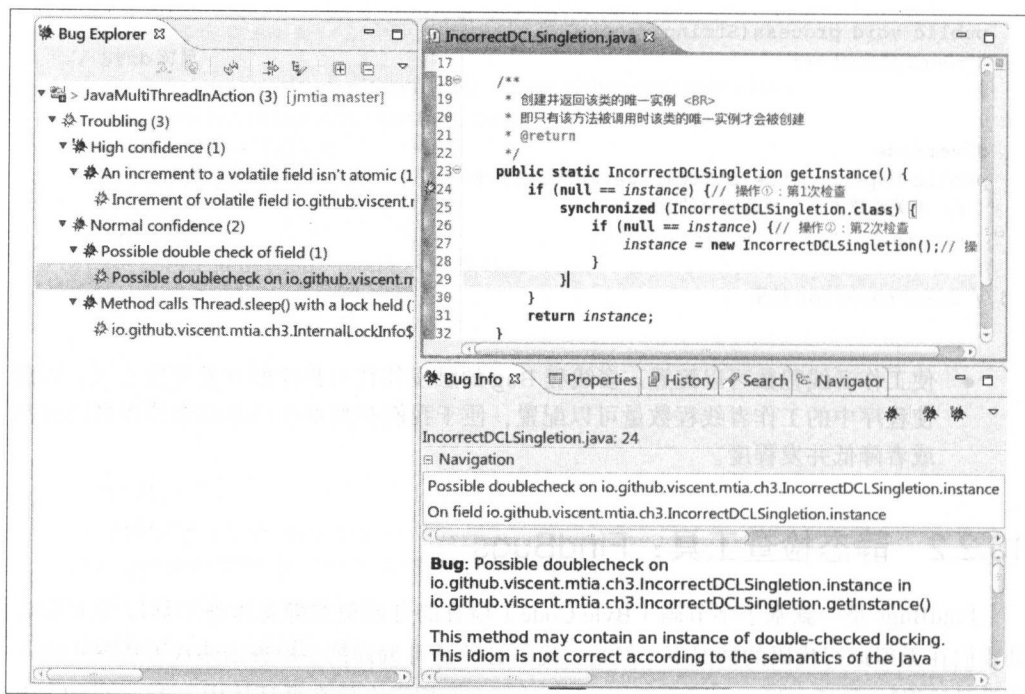


图 10-5 FindBugs 检查结果示例

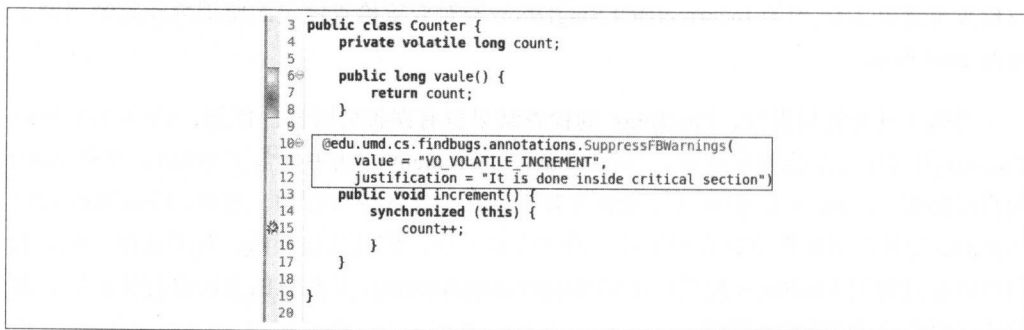


图 10-6 屏蔽 FindBugs 的误报警告

10.2.3 多线程程序的代码复审

代码复审（Code Review）是挖掘多线程程序 Bug 的最有效途径，通过代码复审我们往往能够发现静态检查工具甚至是一些测试途径所不易或者不能发现的问题。然而，代码复审对人的要求也是最高的——代码复审者不仅要具备扎实的多线程编程基础，还需要具备丰富编程经验和良好的代码阅读能力。本着工具能做的事情应该交给工具来做的思想，

代码复审通常是以静态检查通过为前提的，即首先保证静态检查工具发现的“低级错误”得以修复，然后才开始代码复审以避免无谓的人力投入而导致浪费。

下面看几个 FindBugs 目前还不能发现而代码复审可以有效发现的典型多线程 Bug⁶。这些 Bug 往往具有一定的隐蔽性，并且其潜在影响又比较大甚至是致命性的——这好比埋伏的地雷，不踩到它会平安无事，一旦踩到它则往往造成重大伤害！

- `CountDownLatch` 使用的一个典型错误——`CountDownLatch.countDown()`调用没有放在 `finally` 块中，而相应的等待又不是通过调用 `CountDownLatch.await(long timeout, TimeUnit unit)`实现的。这种错误具有一定的隐蔽性——一般性的调试甚至于一些测试手段可能都无法发现这种错误。而一旦 `CountDownLatch.countDown()`调用前的代码抛出异常，那么等待线程可能永远处于等待状态！
- 线程池与网络 I/O 超时时间限制问题。如果线程池执行的任务涉及网络 I/O，那么为这些任务的网络 I/O 操作设置一个合理的等待超时时间限制（包括网络连接超时、网络读取超时等）非常重要。这些网络 I/O 操作如果没有等待超时时间限制，那么极端情况下可能出现线程池中的所有工作者线程都处于无限制的网络等待，从而使得线程池无法接受新提交的任务。这种问题在“正常”的条件下往往很难被发现，而有一定经验的代码复审人员则很容易发现这类问题。
- 用作内部锁句柄的变量未采用 `final` 关键字修饰的问题。这种问题可能导致访问同一组共享数据的多个线程实际上同步在多个内部锁之上，从而造成竞态而违背了使用锁的初衷。例如，Tomcat 早期版本有段代码就有这个问题⁷，如下代码所示：

```
public void addInstanceListener(InstanceListener listener) {
    synchronized (listeners) {
        InstanceListener results[] =
            new InstanceListener[listeners.length + 1];
        for (int i = 0; i < listeners.length; i++)
            results[i] = listeners[i];
        results[listeners.length] = listener;
        listeners = results;
    }
}
```

上述代码中的内部锁句柄对应的是一个数组变量 `listeners`，并且 `listeners` 变量值本身还可以被修改，因此使用 `listeners` 作为内部锁可能导致多个线程同步在不同的数组对象之上！上述代码即便没有重新对 `listeners` 进行赋值，只要 `listeners`

⁶ 指 FindBugs 版本 3.0.1。

⁷ 参见：<http://www.ibm.com/developerworks/cn/java/j-concurrencybugpatterns/>。

变量本身没有采用 `final` 修饰，我们就无法排除后续的代码变更不会出现对 `listeners` 进行赋值的情况。然而，静态检测工具（例如 FindBugs）可能并不能检查出这种错误。

10.2.4 多线程程序的单元测试：JCStress

JCStress 是 OpenJDK 下的一个试验性项目，它可以用来编写多线程程序的单元测试⁸。JCStress 非常直观地体现了多线程程序测试的本质——对特定的共享状态进行并发操作，然后检查实际共享状态（结果）是否符合我们的期望。相应地，JCStress 提供了一组注解（Annotation）和工具类（参见表 10-2），这极大地简化了测试代码编写。

表 10-2 JCStress 常用注解与工具类

注解/类	解 释
@JCStressTest	代表被注释的类是一个 JCStress 测试用例
@State	代表被注释的类包含共享状态
@Actor	代表被注释的方法为并发操作
@Arbiter	相当于一种特殊的 @Actor，其注释的方法会在同一个测试用例内所有 @Actor 注释的并发操作结束后才被执行
@Outcome	代表被注释的测试用例（类）的可能输出结果及其是否可接受
IntResult1、IntResult2、LongResult1、LongResult2 等	这些类代表测试的结果

针对清单 10-3 所示的计数器类 `Counter`，我们可以设计一个简单的测试用例，如表 10-3 所示。

表 10-3 针对类 `Counter` 设计的一个简单测试用例

预置条件	计数器初始值为 0
操作	使用两个并发操作，每个操作执行一次 <code>Counter.increment()</code>
预期	计数器值为 2

清单 10-4 展示了该测试用例对应的 JCStress 代码。

⁸ JCStress 下载地址：<http://openjdk.java.net/projects/code-tools/jcstress/>。

清单 10-3 一个非线程安全的计数器

```

public class Counter {
    private volatile long count;

    public long vaule() {
        return count;
    }

    public void increment() {
        // 此处特意不加锁，以便测试代码能够报告相应的错误
        count++;
    }
}

```

清单 10-4 计数器 Counter 的 JCTest 测试用例

```

@JCStressTest
@Description("测试 Counter 的线程安全性")
@Outcome(id = "[2]", expect = Expect.ACCEPTABLE, desc = "OK")
@Outcome(id = "[1]", expect = Expect.FORBIDDEN, desc = "丢失更新或者读脏数据")
public class CounterTest {
    @State
    public static class StateObject {
        final Counter counter = new Counter();
    }

    @Actor
    public void actor1(StateObject sh) {
        sh.counter.increment();
    }

    @Actor
    public void actor2(StateObject sh) {
        sh.counter.increment();
    }

    @Arbiter
    public void actor3(LongResult1 r, StateObject sh) {
        r.r1 = sh.counter.vaule();
    }
}

```

在上述测试用例中，我们用@State来注解类 StateObject，这表示该类包含了该测试用例所访问的共享状态——测试目标对象 Counter 类的实例。我们用@Actor来注解 actor1、actor2 方法，这表示这些方法要对共享状态进行并发操作。用@Actor注解的方法可以声明类型为代表共享状态的对象的参数。此外，这些方法还可以声明代表测试结果数据（比

如 LongResult1) 的参数用于向 JCTest 报告结果数据。JCTest 会采用一个线程池来执行这些并发操作，并且为了提高触发多线程 Bug 的概率，默认情况下每个并发操作会被执行 5 轮。一个测试用例内所有用 @Actor 注解的方法都被执行一遍算一轮并发操作执行结束，每轮并发操作执行结束之后该测试用例内所有用 @Arbiter 注解的方法就会被执行一次。因此，通常我们会在 @Arbiter 所注解的方法中收集测试结果数据。例如在上述代码中，我们在 actor3 方法中声明了一个 LongResult1 参数用于向 JCTest 提供测试结果数据⁹。由于 JCTest 所提供的表示测试结果的工具类仅支持 int、double、boolean 这类基础数据类型，因此在收集测试结果数据的时候我们可能需要将结果数据转换为基础类型数据。而 JCTest 则根据测试用例中 @Outcome 注解的内容来对测试结果进行解读，即判定并记录相应的结果是否可以接受。

有了上述基础，我们就能够将上述测试用例进一步简化：直接采用 @State 来注解测试用例类本身，如清单 10-5 所示。

清单 10-5 计数器 Counter 的 JCTest 测试用例简化版

```
@JCTestTest
@State
@Description("测试 Counter 的线程安全性")
@Outcome(id = "[2]", expect = Expect.ACCEPTABLE, desc = "OK")
@Outcome(id = "[1]", expect = Expect.FORBIDDEN, desc = "丢失更新或者读脏数据")
public class CounterTestV2 {
    final Counter counter = new Counter();

    @Actor
    public void actor1() {
        counter.increment();
    }

    @Actor
    public void actor2() {
        counter.increment();
    }

    @Arbiter
    public void actor3(LongResult1 r) {
        r.r1 = counter.vaule();
    }
}
```

并发操作多轮执行结束后，JCTest 能够生成测试报告，如图 10-7 所示。从该报告中可以看出某些情况下测试的结果为 1 而不是我们所期望的 2，可见 Counter 类并非线程安全。

9 从 JCTest 的角度来说，这个过程就是收集结果数据。

io.github.viscent.mtia.ch10.CounterTest

测试Counter的线程安全性

Observed state	Occurrence	Expectation	Interpretation
[2]	47858729	ACCEPTABLE	OK
[1]	144191	FORBIDDEN	丢失更新或者读脏数据

图 10-7 Counter 类的测试报告

JCStress 的优点在于其简单性。JCStress 的使用非常简明，使用 JCStress 编写单元测试代码，我们几乎不需要调用 JCStress 的任何 API。这使得测试代码的开发者能够更加专注于测试用例本身的实现而不是与测试工具本身有关的细节以及 API。

JCStress 的缺点表现在以下几个方面。

- 文档的缺乏。JCStress 的相关文档比较少，不过这点一定程度上可以被其简单性所弥补。另外，JCStress 工程的子目录 tests-custom 下有不少针对 JDK 标准库类的测试用例，阅读这些测试用例的源码是学习 JCStress 的一种有效途径。
- 与其他工具的集成。JCStress 目前并没有与 JUnit 集成。JCStress 本身并不提供与 Eclipse 的集成，不过 JCStress 能够以 Maven 项目的形式被集成到 Eclipse 工程之中。本书配套下载资源中的 Maven 工程 (JCStress-tests) 有简明文档 (Readme.doc) 介绍了如何在 Eclipse 中配置和使用 JCStress。
- 不便于测试代码本身的调试。JCStress 并不是直接执行我们所编写的测试用例类，而是执行相应的自动生成的类。例如，针对清单 10-4 中的测试用例 CounterTest 类，JCStress 所执行的是一个名为 CounterTest_jcstress 的类，该类是在 Maven 构建的时候由自动生成的代码自动编译而成的。因此，JCStress 不便于测试代码本身的调试。

选用多线程程序测试工具、框架时的一个重要考量是简单性——测试代码的开发者能够更加专注于测试用例本身的实现而不是与测试工具本身有关的细节以及 API，这正是本书介绍 JCStress 的原因之一。

10.3 本章小结

本章介绍了多线程程序的常用调试技巧以及多线程程序测试的困难性及其常见应对措施。本章知识结构如图 10-8 所示。

在调试过程中使用监视点有助于跟踪线程对共享变量的访问情况。调试多线程程序过程中将断点的暂挂策略设置为暂挂虚拟机策略有助于模拟单线程的调试环境，以便于定位与多线程本身无关的功能性问题；将断点的暂挂策略设置为暂挂线程则有利于定位与多线程有关的非功能性问题。

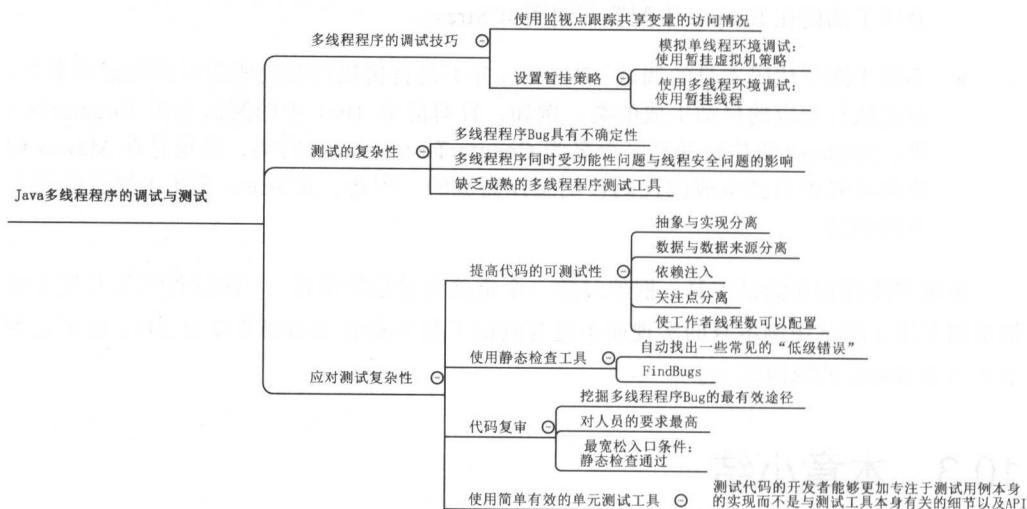
多线程程序测试的复杂性表现为多线程程序的 Bug 具有不确定性、多线程同时受功能性问题与线程安全问题的影响、缺乏成熟的多线程程序测试工具等方面。应对多线程程序测试的困难性可以从提高代码的可测试性、使用静态检查工具、代码复审和选用简单有效的多线程测试工具这几个方面着手。

提高代码的可测试性可以从抽象与实现分离、数据与数据来源分离、依赖注入、关注点分离和使工作者线程数可以配置等几个方面入手。

使用静态检查工具能够帮助我们自动找出一些常见的“低级错误”。

代码复审是挖掘多线程程序 Bug 的最有效途径，但是它对人员的要求也最高。为了减少代码复审的人工成本以提高复审效率，代码复审通常应该在静态检查通过之后进行。

选用多线程程序测试工具、框架时的一个重要考量是简单性——测试代码的开发者能够更加专注于测试用例本身的实现而不是与测试工具本身有关的细节以及 API。



第二部分

多线程编程进阶

→ 第 11 章 多线程编程的硬件基础与 Java 内存模型

→ 第 12 章 Java 多线程程序的性能调校

第 11 章

多线程编程的硬件基础与 Java 内存模型

本章介绍与多线程编程紧密相关的硬件基础知识,这些知识是本书后续章节也是全书的基础知识。本章介绍的许多硬件部件都有这样一个特点——硬件设计者引入一个部件是为了解决某些问题,然而这些部件自身又会引入新的问题。为了解决这些新的问题,硬件设计者又引入了其他部件。因此,掌握这些部件之间的这种关系有助于我们更好地理解相关部件。Java 内存模型是对 Java 多线程程序的正确性进行推理的理论基础,了解 Java 内存模型有助于编写正确的多线程程序以及进行代码复审。

11.1 填补处理器与内存之间的鸿沟：高速缓存

现代处理器处理能力的提升要远胜于主内存 (DRAM) 访问速率的提升,主内存执行一次内存读、写操作所需的时间可能足够处理器执行上百条的指令。为了弥补处理器与主内存处理能力之间的鸿沟,硬件设计者在主内存和处理器之间引入了高速缓存 (Cache),如图 11-1 所示¹。

1 在 SMP (symmetric multiprocessing) 架构下,图中的“互连通道”相当于系统总线 (System Bus)、I/O 桥 (I/O Bridge) 和内存总线 (Memory Bus) 的组合。

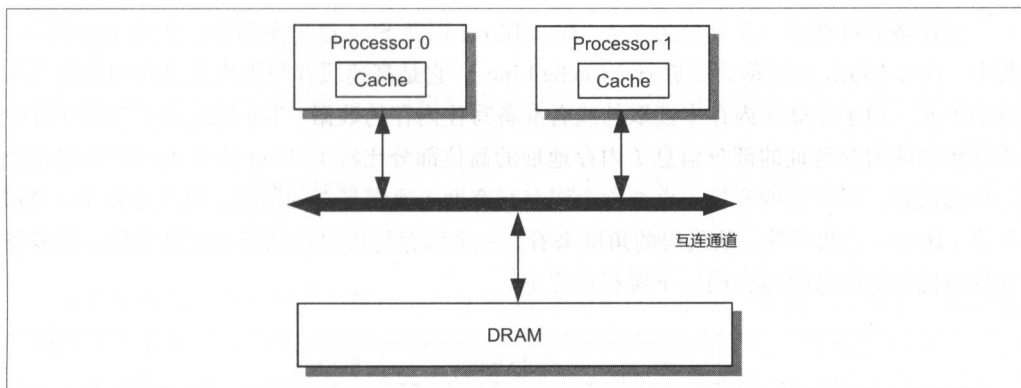


图 11-1 现代计算机系统高速缓存结构

高速缓存是一种存取速率远比主内存大而容量远比主内存小的存储部件，每个处理器都有其高速缓存。引入高速缓存之后，处理器在执行内存读、写操作的时候并不直接与主内存打交道，而是通过高速缓存进行的。变量名相当于内存地址，而变量值则相当于相应内存空间所存储的数据。从这个角度来看，高速缓存相当于为程序所访问的每个变量保留了一份相应内存空间所存储数据（变量值）的副本。由于高速缓存的存储容量远小于主内存，因此高速缓存并不是每时每刻保留着所有变量值的副本。高速缓存相当于一个由硬件实现的容量极小的散列表（Hash Table），其键（Key）是一个内存地址，其值（Value）是内存数据的副本或者准备写入内存的数据。从内部结构来看，高速缓存相当于一个拉链散列表（Chained Hash Table），它包含若干桶（Bucket，硬件上称之为 Set），每个桶又可以包含若干缓存条目（Cache Entry），如图 11-2 所示。

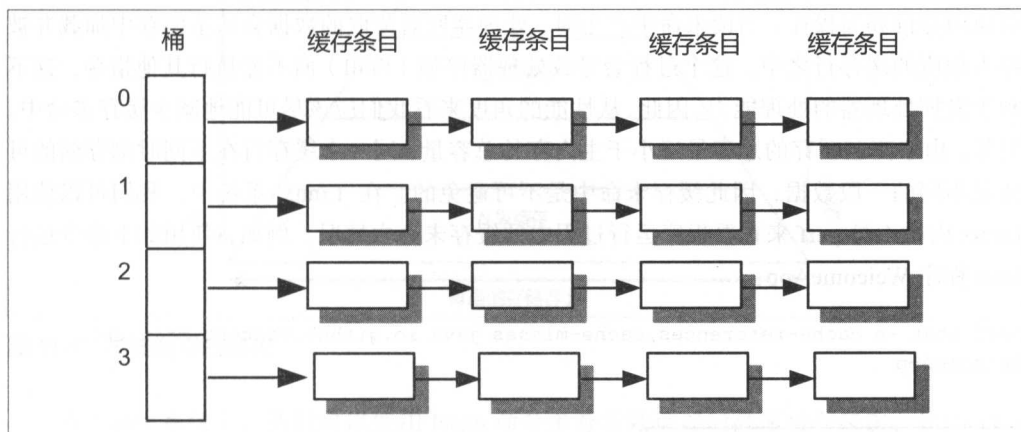


图 11-2 高速缓存内部结构示意图

缓存条目可被进一步划分为 Tag、Data Block 以及 Flag 这三个部分，如图 11-3 所示。其中，Data Block 也被称为缓存行（Cache Line），它是高速缓存与主内存之间的数据交换最小单元，用于存储从内存中读取的或者准备写往内存的数据。Tag 则包含了与缓存行中数据相应的内存地址的部分信息（内存地址的高位部分比特）。Flag 用于表示相应缓存行的状态信息。缓存行的容量（也被称为缓存行宽度）通常是 2 的倍数，其大小在 16~256 字节（Byte）之间不等。从代码的角度来看，一个缓存行可以存储若干变量的值，而多个变量的值则可能被存储在同一个缓存行之中。

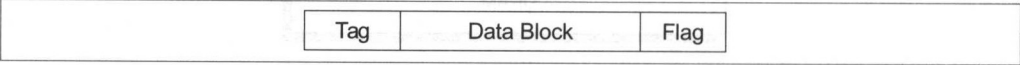


图 11-3 缓存条目的结构

处理器在执行内存访问操作时会将相应的内存地址解码²。内存地址的解码结果包括 tag、index 以及 offset 这三部分数据。其中，index 相当于桶编号，它可以用来定位内存地址对应的桶；一个桶可能包含多个缓存条目，tag 相当于缓存条目的相对编号，其作用在于用来与同一个桶中的各个缓存条目中的 Tag 部分进行比较，以定位一个具体的缓存条目；一个缓存条目中的缓存行可以用来存储多个变量，offset 是缓存行内的位置偏移，其作用在于确定一个变量在一个缓存行中的存储起始位置。根据这个内存地址的解码结果，如果高速缓存子系统能够找到相应的缓存行并且缓存行所在的缓存条目的 Flag 表示相应缓存条目是有效的，那么我们就称相应的内存操作产生了缓存命中（Cache Hit）³；否则，我们就称相应的内存操作产生了缓存未命中（Cache Miss）。

具体来说，缓存未命中包括读未命中（Read Miss）和写未命中（Write Miss），分别对应内存读和写操作。当读未命中产生时，处理器所需读取的数据会从主内存中加载并被存入相应的缓存行之中。这个过程会导致处理器停顿（Stall）而不能执行其他指令，这不利于发挥处理器的处理能力。因此，从性能的角度来看我们应该尽可能地减少缓存未命中。另外，由于高速缓存的总容量远小于主内存的总容量，同一个缓存行在不同时刻存储的可能是不同的一段数据，因此缓存未命中是不可避免的。在 Linux 系统中，我们可以使用 Linux 内核工具 perf 来查看程序运行过程中的缓存未命中情况。例如，使用如下命令运行 Java 程序 WelcomeApp：

```
perf stat -e cache-references,cache-misses java io.github.viscent.mtia.ch1.
WelcomeApp
```

2 具体的解码动作由高速缓存控制器（Cache Controller）负责执行。
3 11.2 节会介绍缓存条目的有效性。

其输出类似如下（省略部分输出）：

```
Performance counter stats for 'java io.github.viscent.mtia.ch1.WelcomeApp':  
  
3,186,985 cache-references  
476,618 cache-misses          # 14.955 % of all cache refs  
  
0.109473235 seconds time elapsed
```

现代处理器一般具有多个层次的高速缓存，如图 11-4 所示。在这个层级中，相应的高速缓存通常被称为一级缓存（L1 Cache）、二级缓存（L2 Cache）、三级缓存（L3 Cache）等。一级缓存可能直接被集成在处理器的内核（Core）里，因此其访问效率非常高，典型的情况是一级缓存的访问操作可以在 2~4 个处理器时钟循环（Clock Cycle）内完成。一级缓存通常包括两部分，其中一部分用于存储指令（L1i），另外一部分用于存储数据（L1d）。距离处理器越近的高速缓存，其存取速率越快，制造成本越高，因此其容量也越小。距离处理器越远（即距离主内存越近）的高速缓存，其存取速率会越慢，而存储容量则相应地增大。

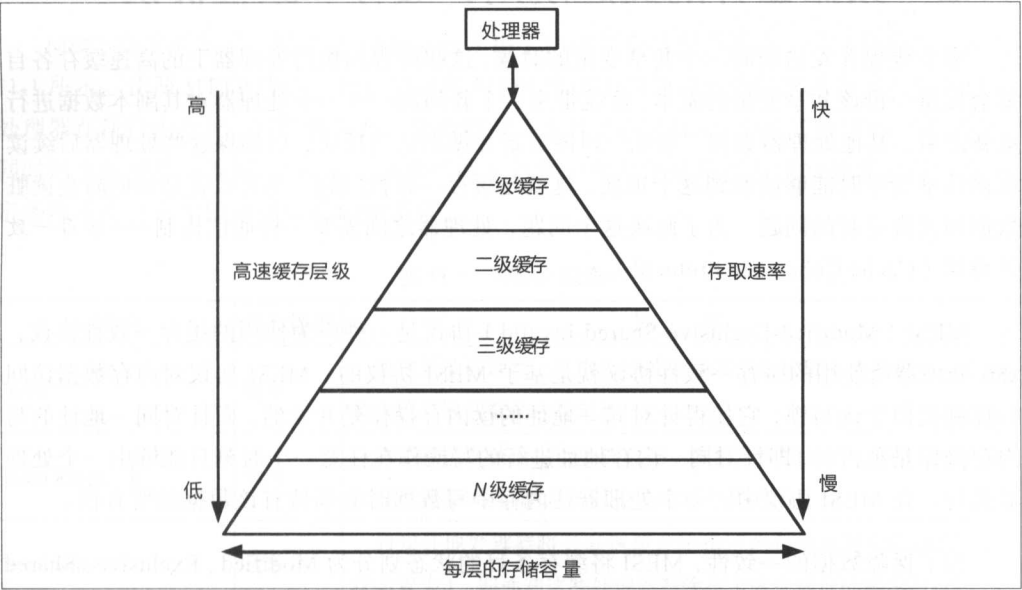


图 11-4 高速缓存的层次

在 Linux 系统下，我们可以使用 `lscpu` 命令来查看处理器的高速缓存层次，如图 11-5 所示。

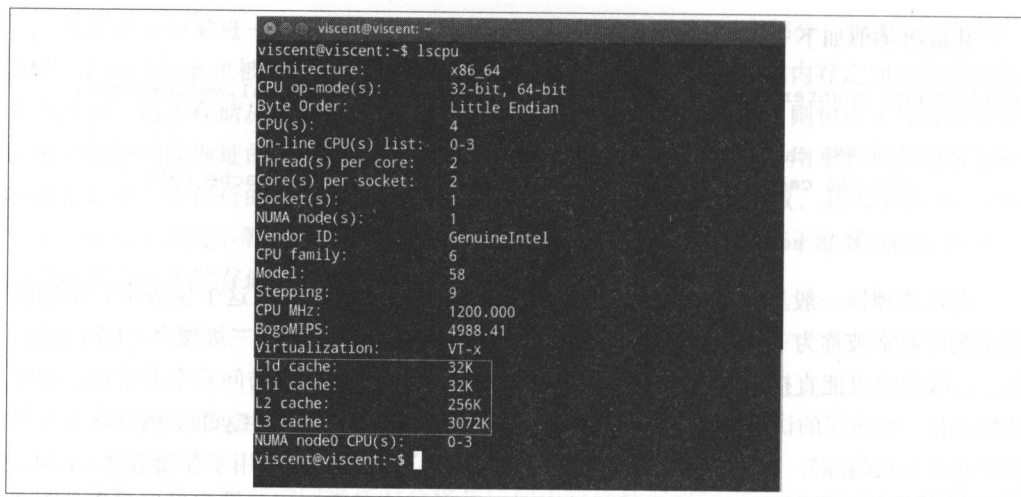


图 11-5 在 Linux 系统中查看高速缓存的层次

11.2 数据世界的交通规则：缓存一致性协议

多个线程并发访问同一个共享变量的时候，这些线程的执行处理器上的高速缓存各自都会保留一份该共享变量的副本，这就带来一个新问题——一个处理器对其副本数据进行更新之后，其他处理器如何“察觉”到该更新并做出适当反应，以确保这些处理器后续读取该共享变量时能够读取到这个更新。这就是缓存一致性问题，其实质就是如何防止脏数据和丢失更新的问题。为了解决这个问题，处理器之间需要一种通信机制——缓存一致性协议（Cache Coherence Protocol）。

MESI（Modified-Exclusive-Shared-Invalid）协议是一种广为使用的缓存一致性协议，x86 处理器所使用的缓存一致性协议就是基于 MESI 协议的。MESI 协议对内存数据访问的控制类似于读写锁，它使得针对同一地址的读内存操作是并发的，而针对同一地址的写内存操作是独占的，即针对同一内存地址进行的写操作在任意一个时刻只能由一个处理器执行。在 MESI 协议中，一个处理器往内存中写数据时必须持有该数据的所有权。

为了保障数据的一致性，MESI 将缓存条目的状态划分为 Modified、Exclusive、Shared 和 Invalid 这 4 种，并在此基础上定义了一组消息（Message）用于协调各个处理器的读、写内存操作。

MESI 协议中一个缓存条目的 Flag 值有以下 4 种可能。

- Invalid（无效的，记为 I）。该状态表示相应缓存行中不包含任何内存地址对应的

有效副本数据。该状态是缓存条目的初始状态。

- **Shared**（共享的，记为 S）。该状态表示相应缓存行包含相应内存地址所对应的副本数据。并且，其他处理器上的高速缓存中也可能包含相同内存地址对应的副本数据。因此，一个缓存条目的状态如果为 Shared，并且其他处理器上也存在 Tag 值与该缓存条目的 Tag 值相同的缓存条目，那么这些缓存条目的状态也为 Shared。处于该状态的缓存条目，其缓存行中包含的数据与主内存中包含的数据一致。
- **Exclusive**（独占的，记为 E）。该状态表示相应缓存行包含相应内存地址所对应的副本数据。并且，该缓存行以独占的方式保留了相应内存地址的副本数据，即其他所有处理器上的高速缓存当前都不保留该数据的有效副本。处于该状态的缓存条目，其缓存行中包含的数据与主内存中包含的数据一致。
- **Modified**（更改过的，记为 M）。该状态表示相应缓存行包含对相应内存地址所做的更新结果数据。由于 MESI 协议中的任意一个时刻只能有一个处理器对同一内存地址对应的数据进行更新，因此在多个处理器上的高速缓存中 Tag 值相同的缓存条目中，任意一个时刻只能有一个缓存条目处于该状态。处于该状态的缓存条目，其缓存行中包含的数据与主内存中包含的数据不一致。

MESI 协议定义了一组消息（Message）用于协调各个处理器的读、写内存操作，如表 11-1 所示。比照 HTTP 协议，我们可以将 MESI 协议中的消息分为请求消息和响应消息。处理器在执行内存读、写操作时在必要的情况下会往总线（Bus）中发送特定的请求消息，同时每个处理器还嗅探（Snoop，也称拦截）总线中由其他处理器发出的请求消息并在一定条件下往总线中回复相应的响应消息。

表 11-1 MESI 消息

消息名	消息类型	描 述
Read	请求	通知其他处理器、主内存当前处理器准备读取某个数据。该消息包含待读取数据的内存地址
Read Response	响应	该消息包含被请求读取的数据。该消息可能是主内存提供的，也可能是嗅探 Read 消息的其他高速缓存提供的
Invalidate	请求	通知其他处理器将其高速缓存中指定内存地址对应的缓存条目状态置为 I，即通知这些处理器删除指定内存地址的副本数据 ⁴
Invalidate Acknowledge	响应	接收到 Invalidate 消息的处理器必须回复该消息，以表示删除了其高速缓存上的相应副本数据

4 这里的删除指逻辑意义上的删除，实际的实现是通过更新相应缓存条目的 Flag 值。

续表

消息名	消息类型	描 述
Read Invalidate	请求	该消息是由 Read 消息和 Invalidate 消息组合而成的复合消息。其作用在于通知其他处理器当前处理器准备更新（Read-Modify-Write，读后写更新）一个数据，并请求其他处理器删除其高速缓存中相应的副本数据。接收到该消息的处理器必须回复 Read Response 消息和 Invalidate Acknowledge 消息
Writeback	请求	该消息包含需要写入主内存的数据及其对应的内存地址

下面看看使用 MESI 协议的处理器是如何实现内存读、写操作的。假设内存地址 A 上的数据 S 是处理器 Processor 0 和处理器 Processor 1 可能共享的数据。

下面讨论在 Processor 0 上读取数据 S 的实现。Processor 0 会根据地址 A 找到对应的缓存条目，并读取该缓存条目的 Tag 和 Flag 值（缓存条目状态）。为讨论方便，这里我们不讨论 Tag 值的匹配问题。Processor 0 找到的缓存条目的状态如果为 M、E 或者 S，那么该处理器可以直接从相应的缓存行中读取地址 A 所对应的数据，而无须往总线中发送任何消息。Processor 0 找到的缓存条目的状态如果为 I，则说明该处理器的高速缓存中并不包含 S 的有效副本数据，此时 Processor 0 需要往总线发送 Read 消息以读取地址 A 对应的数据，而其他处理器 Processor 1（或者主内存）则需要回复 Read Response 以提供相应的数据，如表 11-2 所示。

表 11-2 处理器对共享数据读操作的实现

		Processor 0	Processor 1
缓存条目	当前状态	I	M/E/S
	目标状态	S	S
消 息	发送	Read 消息	Read Response 消息
	接收	Read Response 消息	Read 消息

Processor 0 接收到 Read Response 消息时，会将其中携带的数据（包含数据 S 的数据块）存入相应的缓存行并将相应缓存条目的状态更新为 S。Processor 0 接收到的 Read Response 消息可能来自主内存也可能来自其他处理器（Processor 1）。Processor 1 会嗅探总线中由其他处理器发送的消息。Processor 1 嗅探到 Read 消息的时候，会从该消息中取出待读取的内存地址，并根据该地址在其高速缓存中查找对应的缓存条目。如果 Processor 1 找到的缓存条目的状态不为 I（表 11-2 所示的情况），则说明该处理器的高速缓存中有待读取数据的副本，此时 Processor 1 会构造相应的 Read Response 消息并将相应缓存行所存

储的整块数据（而不仅仅是 Processor 0 所请求的数据 S）“塞入”该消息。如果 Processor 1 找到的相应缓存条目的状态为 M，那么 Processor 1 可能在往总线发送 Read Response 消息前将相应缓存行中的数据写入主内存。Processor 1 往总线发送 Read Response 之后，相应缓存条目的状态会被更新为 S。如果 Processor 1 找到的高速缓存条目的状态为 I，那么 Processor 0 所接收到的 Read Response 消息就来自主内存。可见，在 Processor 0 读取内存的时候，即便 Processor 1 对相应的内存数据进行了更新且这种更新还停留在 Processor 1 的高速缓存中而造成高速缓存与主内存中的数据不一致，在 MESI 消息的协调下这种不一致也并不会导致 Processor 0 读取到一个过时的旧值。

下面讨论 Processor 0 往地址 A 写数据的实现。任何一个处理器执行内存写操作时必须拥有相应数据的所有权。在执行内存写操作时，Processor 0 会先根据内存地址 A 找到相应的缓存条目。Processor 0 所找到的缓存条目的状态若为 E 或者 M，则说明该处理器已经拥有相应数据的所有权，此时该处理器可以直接将数据写入相应的缓存行并将相应缓存条目的状态更新为 M⁵。Processor 0 所找到的缓存条目的状态如果不为 E、M，则该处理器需要往总线发送 Invalidate 消息以获得数据的所有权。其他处理器接收到 Invalidate 消息后会将其高速缓存中相应的缓存条目状态更新为 I（相当于删除相应的副本数据）并回复 Invalidate Acknowledge 消息。发送 Invalidate 消息的处理器（即内存写操作的执行处理器），必须在接收到其他所有处理器所回复的所有 Invalidate Acknowledge 消息之后再将数据更新到相应的缓存行之中，如表 11-3 所示。

表 11-3 处理器对共享数据写操作的实现

		Processor 0		Processor 1	
		场景 1	场景 2	场景 1	场景 2
缓存条目	当前状态	S	I	I	M/E/S
	目标状态	M	M	I	I
消 息	发送	Invalidate	Read Invalidate	Invalidate Acknowledge	Read Response 和 Invalidate Acknowledge
	接收	Invalidate Acknowledge	Read Response 和 Invalidate Acknowledge	Invalidate	Read Invalidate

Processor 0 所找到的缓存条目的状态若为 S，则说明 Processor 1 上的高速缓存可能也保留了地址 A 对应的数据副本（场景 1），此时 Processor 0 需要往总线发送 Invalidate 消息。Processor 0 在接收到其他所有处理器所回复的 Invalidate Acknowledge 消息之后会将

5 如果本来状态就为 M，则无须更新。

相应的缓存条目的状态更新为 E，此时 Processor 0 获得了地址 A 上数据的所有权。接着，Processor 0 便可以将数据写入相应的缓存行，并将相应的缓存条目的状态更新为 M。Processor 0 所找到的缓存条目的状态若为 I，则表示该处理器不包含地址 A 对应的有效副本数据（场景 2），此时 Processor 0 需要往总线发送 Read Invalidate 消息。Processor 0 在接收到 Read Response 消息以及其他所有处理器所回复的 Invalidate Acknowledge 消息之后，会将相应缓存条目的状态更新为 E，这表示该处理器已经获得相应数据的所有权。接着，Processor 0 便可以往相应的缓存行中写入数据了并将相应缓存条目的状态更新为 M。其他处理器在接收到 Invalidate 消息或者 Read Invalidate 消息之后，必须根据消息中包含的内存地址在该处理器的高速缓存中查找相应的高速缓存条目。若 Processor 1 所找到的高速缓存条目的状态不为 I（场景 2），那么 Processor 1 必须将相应缓存条目的状态更新为 I，以删除相应的副本数据并给总线回复 Invalidate Acknowledge 消息。可见，Invalidate 消息和 Invalidate Acknowledge 消息使得针对同一个内存地址的写操作在任意一个时刻只能由一个处理器执行，从而避免了多个处理器同时更新同一数据可能导致的数据不一致问题。

从上述例子来看，在多个线程共享变量的情况下，MESI 协议已经能够保障一个线程对共享变量的更新对其他处理器上运行的线程来说是可见的；既然如此，第 2 章中所说的可见性又何以存在呢？这就要从写缓冲器和无效化队列的角度来解释了。

11.3 硬件缓冲器：写缓冲器与无效化队列

MESI 协议解决了缓存一致性问题，但是其自身也存在一个性能弱点——处理器执行写内存操作时，必须等待其他所有处理器将其高速缓存中的相应副本数据删除并接收到这些处理器所回复的 Invalidate Acknowledge/Read Response 消息之后才能将数据写入高速缓存。为了规避和减少这种等待造成的写操作的延迟（Latency），硬件设计者引入了写缓冲器和无效化队列，如图 11-6 所示。

写缓冲器（Store Buffer，也被称为 Write Buffer）是处理器内部的一个容量比高速缓存还小的私有高速存储部件⁶，每个处理器都有其写缓冲器，写缓冲器内部可包含若干条目（Entry）。一个处理器无法读取另外一个处理器上的写缓冲器中的内容。

6 写缓冲器的容量大小通常等于一级缓存的缓存行宽度，比如 32 字节、64 字节。

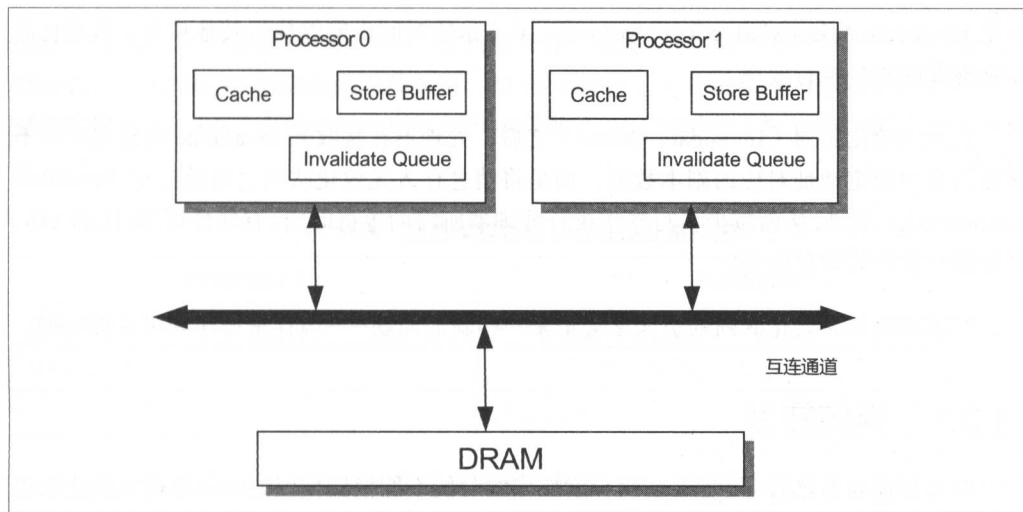


图 11-6 引入写缓冲器和无效化队列的处理器

引入写缓冲器之后，处理器在执行写操作时会做这样的处理：如果相应的缓存条目状态为 E 或者 M，那么处理器可能会直接将数据写入相应的缓存行而无须发送任何消息⁷；如果相应的缓存条目状态为 S，那么处理器会先将写操作的相关数据（包括数据和待操作的内存地址）存入写缓冲器的条目之中，并发送 Invalidate 消息；如果相应的缓存条目状态为 I，我们就称相应的写操作遇到了写未命中（Write Miss），那么此时处理器会先将写操作相关数据存入写缓冲器的条目之中，并发送 Read Invalidate 消息。我们知道在其他所有处理器的高速缓存都未保存指定地址的副本数据的情况下，Read 消息回复者是主内存，也就是说 Read 消息可能导致内存读操作。因此，写未命中的开销是比较大的。内存写操作的执行处理器在将写操作的相关数据写入写缓冲器之后便认为该写操作已经完成，即该处理器并不等待其他处理器返回 Invalidate Acknowledge/Read Response 消息而是继续执行其他指令（比如执行读操作）。一个处理器接收到其他处理器所回复的针对同一个缓存条目的所有 Invalidate Acknowledge 消息的时候，该处理器会将写缓冲器中针对相应地址的写操作的结果写入相应的缓存行中，此时写操作对于其执行处理器之外的其他处理器来说才算是完成的。

由此可见，写缓冲器的引入使得处理器在执行写操作的时候可以不等等待 Invalidate Acknowledge 消息，从而减少了写操作的延时，这使得写操作的执行处理器在其他处理器

7 取决于具体处理器的实现。也有的处理器（比如 x86 处理器）不管相应的缓存条目状态如何，直接先将每一个写操作的结果都存入写缓冲器。

回复 Invalidate Acknowledge/Read Response 消息这段时间内能够执行其他指令，从而提高了处理器的指令执行效率。

引入无效化队列（Invalidate Queue）之后，处理器在接收到 Invalidate 消息之后并不删除消息中指定地址对应的副本数据，而是将消息存入无效化队列之后就回复 Invalidate Acknowledge 消息，从而减少了写操作执行处理器所需的等待时间。有些处理器（比如 x86）可能没有使用无效化队列。

写缓冲器和无效化队列的引入又会带来一些新的问题——内存重排序和可见性问题。

11.3.1 存储转发

引入写缓冲器之后，处理器在执行读操作的时候不能根据相应的内存地址直接读取相应缓存行中的数据作为该操作的结果。这是因为一个处理器在更新一个变量之后紧接着又读取该变量的值的时候，由于该处理器先前对该变量的更新结果可能仍然还停留在写缓冲器之中，因此该变量相应的内存地址所对应的缓存行中存储的值是该变量的旧值。这种情况下为了避免读操作所返回的结果是一个旧值，处理器在执行读操作的时候会根据相应的内存地址查询写缓冲器。如果写缓冲器存在相应的条目，那么该条目所代表的写操作的结果数据就会直接作为该读操作的结果返回；否则，处理器才从高速缓存中读取数据。这种处理器直接从写缓冲器中读取数据来实现内存读操作的技术被称为存储转发（Store Forwarding）。存储转发使得写操作的执行处理器能够在不影响该处理器执行读操作的情况下将写操作的结果存入写缓冲器。

11.3.2 再探内存重排序

写缓冲器和无效化队列都可能导致内存重排序。

写缓冲器可能导致 StoreLoad 重排序（Stores Reordered After Loads）。StoreLoad 重排序是绝大多数处理器都允许的一种内存重排序。假设处理器 Processor 0 和 Processor 1 上的两个线程未使用任何同步措施而各自按照程序顺序并依照表 11-4 所示的线程交错顺序执行。其中变量 X、Y 为共享变量，其初始值均为 0，r1、r2 为局部变量。当 Processor 0 上的线程执行到 L2 时，虽然在此之前 S3 已经被 Processor 1 执行完毕，但是由于 S3 的执行结果可能仍然还停留在 Processor 1 的写缓冲器中，而一个处理器无法读取另外一个处理器的写缓冲器中的内容，因此 Processor 0 此刻读取到的 Y 的值仍然是其高速缓存中存储的该变量的初始值 0。同理，Processor 1 执行到 L4 时所读取到变量 X 的值也可能是该

变量的初始值 0。因此，从 Processor 1 的角度来看，Processor 1 执行 L4 的那一刻 Processor 0 已经执行了 L2 而 S1 却像是尚未被执行，即 Processor 1 对 Processor 0 执行的两个操作的感知顺序是 L2→S1，也就是说此时写缓冲器导致了 S1（写操作）被重排序到了 L2（读操作）之后。

表 11-4 写缓冲器导致 StoreLoad 重排序

Processor 0	Processor 1
X=1;//S1	Y=1;//S3
r1=Y;//L2	
	r2=X;//L4

StoreLoad 重排序可能导致某些算法失效。例如，Peterson 算法中两个线程的操作序列与表 11-4 类似，因此 StoreLoad 重排序就可能导致该算法失效⁸。

写缓冲器可能导致 StoreStore 重排序（Stores Reordered After Stores）。假设处理器 Processor 0 和 Processor 1 上的两个线程未使用任何同步措施而各自按照程序顺序并依照表 11-5 所示的线程交错顺序执行。其中变量 data、ready 为共享变量，其初始值分别为 0 和 false。假设 Processor 0 执行 S1、S2 时该处理器的高速缓存中包含变量 ready 的副本但不包含变量 data 的副本，那么 S1 的执行结果会先被存入写缓冲器而 S2 的执行结果会直接被存入高速缓存。L3 被执行时 S2 对 ready 的更新通过缓存一致性协议可以被 Processor 1 读取到，于是，由于 ready 值已变为 true，因此 Processor 1 继续执行 L4。L4 被执行的时候，由于 S1 对 data 的更新结果可能仍然停留在 Processor 0 的写缓冲器之中，因此 Processor 1 此时读取到的变量 data 的值可能仍然是其初始值 0，即 L4 的输出结果可能仍然是 0 而不是 Processor 1 所期望的新值（Processor 0 更新之后的值）。从 Processor 1 的角度来看，这就造成了一种现象——S2 像是先于 S1 被执行，即 S1（写操作）被重排序（内存重排序）到了 S2（写操作）之后。同样，StoreStore 重排序也可能导致某些算法失效。

表 11-5 写缓冲器导致 StoreStore 重排序

Processor 0	Processor 1
data=1;//S1	
ready=true;//S2	
	while(!ready) continue;//L3
	print(data);//L4

8 参见：https://en.wikipedia.org/wiki/Peterson's_algorithm。

另外，某些处理器（比如 ARM 处理器）为了充分利用总线带宽（Bus Bandwidth）以提高将写缓冲器中的内容冲刷（写入）到高速缓存的效率，会将针对连续内存地址的写操作⁹并入同一个写缓冲器条目之中，这种处理就被称为写合并（Write Combining）。写合并也可能导致 StoreStore 重排序。

无效化队列可能导致 LoadLoad 重排序（Loads Reordered After Loads）。假设处理器 Processor 0 和 Processor 1 上的两个线程未使用任何同步措施而各自按照程序顺序并依照表 11-5 所示的线程交错顺序执行。其中变量 data、ready 为共享变量，其初始值分别为 0 和 false。进一步假设 Processor 0 的高速缓存中存有变量 data 和 ready 的副本，Processor 1 仅存有变量 data 的副本而未存有变量 ready 的副本。那么，Processor 0 和 Processor 1 有可能按照如下序列执行一系列操作。

① Processor 0 执行 S1。此时由于 Processor 1 上也存有变量 data 的副本，因此 Processor 0 会发出 Invalidate 消息并将 S1 的操作结果存入写缓冲器。

② Processor 1 接收到 Processor 0 发出的 Invalidate 消息时将该消息存入其无效化队列并回复 Invalidate Acknowledge 消息。

③ Processor 0 接收到 Invalidate Acknowledge 消息，随即将 S1 的操作结果写入高速缓存。然后，Processor 0 执行 S2。此时由于只有 Processor 0 上存有变量 ready 的副本，因此 Processor 0 无须发送任何消息，直接将 S2 的操作结果存入高速缓存即可。

④ Processor 1 执行 L3。此时由于 Processor 1 的高速缓存中并没有存储变量 ready 的副本，因此 Processor 1 会发出一个 Read 消息。

⑤ Processor 0 接收到 Processor 1 发出的 Read 消息并回复 Read Response 消息。由于此时 Processor 0 已经执行过 S2，因此该 Read Response 消息包含的 ready 变量值为 true。

⑥ Processor 1 接收到 Read Response 消息并从中取出 ready 变量的新值（true），此时 L3 中的循环语句可以结束。

⑦ Processor 1 执行 L4。此时，由于 Processor 0 为了更新变量 data 而发出的 Invalidate 消息可能仍然还停留在 Processor 1 的无效化队列中，因此 Processor 1 从其高速缓存中读取的变量 data 的值仍然是其初始值。因此，L4 所打印的变量值可能是一个旧值。

由此可见，尽管 Processor 0 对共享变量 data、ready 的更新是按照程序顺序先后到达

⁹ 这些写操作在时间上可以是不连续的。

高速缓存的,但是由于无效化队列的作用 Processor 1 像是在 ready 变量不为 true 的情况下提前读取了变量 data 的值,然而,程序的实际处理逻辑是仅在 ready 变量值为 true 的情况下才读取变量 data,因此这里 Processor 1 实际读取到的变量 (data) 值是一个旧值。也就是说,从 Processor 0 的角度来看, L4 (读操作) 被重排序到了 L3 (读操作) 之前。可见, LoadLoad 重排序会导致类似 StoreStore 重排序的效果。

不同的处理器架构所支持 (允许) 的内存重排序各有不同。比如,现代处理器都会采用写缓冲器,而有的处理器 (比如 x86) 会保障写操作的顺序,即这些处理器不允许 StoreStore 重排序的出现。

11.3.3 再探可见性

写缓冲器是处理器内部的私有存储部件,一个处理器中的写缓冲器所存储的内容是无法被其他处理器所读取的。因此,一个处理器上运行的线程更新了一个共享变量之后,其他处理器上运行的线程再来读取该变量时这些线程可能仍然无法读取到前一个线程对该变量所做的更新,因为这个更新可能还停留在前一个线程所在的处理器上的写缓冲器之中。这种现象就是前面章节所说的可见性问题。因此,我们说写缓冲器是可见性问题的硬件根源¹⁰。为了使一个处理器上运行的线程对共享变量所做的更新可以被其他处理器上运行的其他线程所读取,我们必须将写缓冲器中的内容写入其所在的处理器上的高速缓存之中,从而使该更新在缓存一致性协议的作用下可以被其他处理器读取到。实现这一点就是前面章节所说的保证一个处理器上运行的线程对共享变量所做的更新可以被其他处理器 (及其上运行的线程) 同步。处理器在一些特定条件下 (比如写缓冲器满、I/O 指令被执行) 会将写缓冲器排空 (Drain) 或者冲刷 (Flush)¹¹,即将写缓冲器中的内容写入高速缓存,但是从程序对一个或者一组变量更新的角度来看,处理器本身并无法保证这种冲刷对程序来说是“及时”的。因此,为了保证一个处理器对共享变量所做的更新可以被其他处理器同步,编译器等底层系统需要借助一类被称为内存屏障的特殊指令。内存屏障中的存储屏障 (Store Barrier) 可以使执行该指令的处理器冲刷其写缓冲器。

然而,冲刷写缓冲器只是解决了可见性问题的一半。因为可见性问题的另一半是无效化队列导致的。无效化队列的引入本身也会导致新的问题——处理器在执行内存读取操作前如果没有根据无效化队列中的内容将该处理器上的高速缓存中的相关副本数据删除,那

10 前面章节我们提到过编译器的优化也可能导致可见性问题。

11 参见 Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A 的“11.10 STORE BUFFER”。

么就可能导致该处理器读到的数据是过时的旧数据，从而使得其他处理器所做的更新丢失。因此，为了使一个处理器上运行的线程能够读取到另外一个处理器上运行的线程对共享变量所做的更新，该处理器必须先根据无效化队列中存储的 Invalidate 消息删除其高速缓存中的相应副本数据，从而使其他处理器上运行的线程对共享变量所做的更新在缓存一致性协议的作用下能够被同步到该处理器的高速缓存之中。内存屏障中的加载屏障（Load Barrier）正是用来解决这个问题的。加载屏障会根据无效化队列内容所指定的内存地址，将相应处理器上的高速缓存中相应的缓存条目的状态都标记为 I，从而使该处理器后续执行针对相应地址（无效化队列内容中指定的地址）的读内存操作时必须发送 Read 消息，以将其他处理器对相关共享变量所做的更新同步到该处理器的高速缓存中。

因此，解决可见性问题首先要使写线程对共享变量所做的更新能够到达（被存储到）高速缓存，从而使该更新对其他处理器是可同步的。其次，读线程所在的处理器要将其无效化队列中的内容“应用”到其高速缓存上，这样才能够将其他处理器对共享变量所做的更新同步到该处理器的高速缓存中。而这两点是通过存储屏障与加载屏障的成对使用实现的：写线程的执行处理器所执行的存储屏障保障了该线程对共享变量所做的更新对读线程来说是可同步的；读线程的执行处理器所执行的加载屏障将写线程对共享变量所做的更新同步到该处理器的高速缓存之中。

存储转发技术也可能导致可见性问题。假设处理器 Processor 0 在 t_1 时刻更新了某个共享变量，随后又在 t_2 时刻读取了该变量。在 t_1 时刻到 t_2 时刻之间的这段时间内其他处理器可能已经更新了该共享变量，并且这个更新的结果已经到达该处理器的高速缓存。但是如果 Processor 0 在 t_1 时刻所做的更新仍然停留在该处理器的写缓冲器之中，那么存储转发技术会使 Processor 0 直接从其写缓冲器读取该共享变量的值。也就是说 Processor 0 此时根本不从高速缓存中读取该变量的值，这就使得另外一个处理器对该共享变量所做的更新无法被该处理器读取，从而导致 Processor 0 在 t_2 时刻读取到的变量值是一个旧值。因此，考虑到存储转发技术的这个副作用，从读线程的角度来看，为了使读线程能够将其他线程对共享变量所做的更新同步到该线程所在的处理器的高速缓存中，我们需要清空该处理器上的写缓冲器以及无效化队列。

11.4 基本内存屏障

处理器支持哪种内存重排序（LoadLoad 重排序、LoadStore 重排序、StoreStore 重排序和 StoreLoad 重排序），就会提供能够禁止相应重排序的指令，这些指令就被称为基本内存屏障——LoadLoad 屏障、LoadStore 屏障、StoreStore 屏障和 StoreLoad 屏障。基本内存

屏障可以统一用 XY 来表示，其中的 X 和 Y 可以代表 Load 或者 Store。基本内存屏障是对一类指令的称呼，这类指令的作用是禁止该指令左侧的任何 X 操作与该指令右侧的任何 Y 操作之间进行重排序，从而确保该指令左侧的所有 X 操作先于该指令右侧的 Y 操作被提交，即内存操作作用到主内存（或者高速缓存）上，如表 11-6 所示。比如，StoreLoad 屏障（即 X 代表 Store，Y 代表 Load）能够禁止其左侧的任何写操作与其右侧的任何读操作之间进行重排序，因此 StoreLoad 屏障就保障了该指令之前的写操作的结果在该指令之后的任何读操作的数据被加载之前对其他处理器来说可同步，即这些写操作的结果会在该屏障之后的读操作的数据被加载前被写入高速缓存（或者主内存）。

表 11-6 基本内存屏障的具体作用

屏障名称	示例指令序列	具体作用
StoreLoad	Store1;Store2;Store3; StoreLoad ; Load1;Load2;Load3	禁止 StoreLoad 重排序,即确保该屏障之前的任何一个写操作（比如 Store2）的结果都会在该屏障之后的任何一个读操作（比如 Load1）的数据被加载之前对其他处理器来说是可同步的
StoreStore	Store1;Store2;Store3; StoreStore ; Store4;Store5;Store6	禁止 StoreStore 重排序,即确保该屏障之前的任何一个写操作（比如 Store1）的结果都会在该屏障之后的任何一个写操作（比如 Store4）之前对其他处理器来说是可同步的
LoadLoad	Load1;Load2;Load3; LoadLoad ; Load4;Load5;Load6	禁止 LoadLoad 重排序,即确保该屏障之前的任何一个读操作（比如 Load1）的数据都会在该屏障之后的任何一个读操作（比如 Load4）之前被加载
LoadStore	Load1;Load2;Load3; LoadStore ; Store1; Store2; Store3	禁止 LoadStore 重排序,即确保该屏障之前的任何一个读操作（比如 Load1）的数据都会在该屏障之后的任何一个写操作（比如 Store1）的结果被冲刷（写入）到高速缓存（或者主内存）之前被加载

基本内存屏障的作用只是保障其左侧的 X 操作（比如读，即 X 代表 Load）先于其右侧的 Y 操作（比如写，即 Y 代表 Store）被提交，它并不全面禁止重排序。XY 屏障两侧的内存操作仍然可以在不越过内存屏障本身的情况下在各自的范围内进行重排序，并且 XY 屏障左侧的非 X 操作与屏障右侧的非 Y 操作之间仍然可以进行重排序（即越过内存屏障本身）。例如，在图 11-7 所示的指令序列中 Load2、Load3 和 Store1、Store2 之间无法进行重排序，而 Store1、Load1 和 Store2 之间可以重排序，Store3、Load2 和 Load3 之间可以重排序，Load1 和 Store3 之间也可以进行重排序。

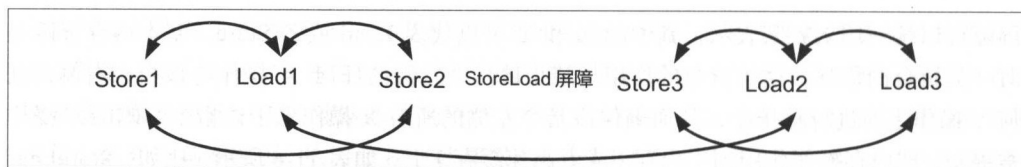


图 11-7 StoreLoad 屏障允许的重排序示意图

编译器（JIT 编译器）、运行时（Java 虚拟机）和处理器都会尊重内存屏障，从而保障其作用得以落实。例如，在图 11-7 所示的指令序列中，假如编译器不尊重内存屏障而在其动态编译（JIT 编译）的时候将 Store1 重排序（指令重排序）到 Load2 之后，那么显然 StoreLoad 屏障被架空了：它无法保障 Store1 先于 Load2 被提交。因此，内存屏障需要得到编译器等多方的尊重，其作用才能落实。

LoadLoad 屏障是通过清空无效化队列来实现禁止 LoadLoad 重排序的。LoadLoad 屏障会使其执行处理器根据无效化队列中的 Invalidate 消息删除其高速缓存中相应的副本。这个过程被称为将无效化队列应用到高速缓存，也被称为清空无效化队列，它使处理器有机会将其他处理器对共享变量所做的更新同步到该处理器的高速缓存中，从而消除了 LoadLoad 重排序的根源而实现了禁止 LoadLoad 重排序¹²。

StoreStore 屏障可以通过对写缓冲器中的条目进行标记来实现禁止 StoreStore 重排序。StoreStore 屏障会将写缓冲器中的现有条目做一个标记，以表示这些条目代表的写操作需要先于该屏障之后的写操作被提交。处理器在执行写操作的时候如果发现写缓冲器中存在被标记的条目，那么即使这个写操作对应的高速缓存条目的状态为 E 或者 M，此时处理器也不直接将写操作的数据写入高速缓存，而是将其写入写缓冲器，从而使得 StoreStore 屏障之前的任何写操作先于该屏障之后的写操作被提交。

就处理器的具体实现而言，许多处理器往往将 StoreLoad 屏障实现为一个通用基本内存屏障（General-purpose Barrier），即 StoreLoad 屏障能够实现其他 3 种基本内存屏障的效果。StoreLoad 屏障能够替代其他基本内存屏障，但是它的开销也是最大的——StoreLoad 屏障会清空无效化队列，并将写缓冲器中的条目冲刷（写入）高速缓存。因此，StoreLoad 屏障既可以将其他处理器对共享变量所做的更新同步到该处理器的高速缓存中，又可以使其执行处理器对共享变量所做的共享对其他处理器来说可同步。

¹² 这里有个前提是，其他处理器对共享变量所做的更新必须到达这些处理器的高速缓存之中。

11.5 Java 同步机制与内存屏障

Java 虚拟机对 `synchronized`、`volatile` 和 `final` 关键字的语义的实现就是借助内存屏障的。第 3 章介绍的获取屏障和释放屏障相当于由基本内存屏障组合而成的复合屏障。获取屏障相当于 `LoadLoad` 屏障和 `LoadStore` 屏障的组合，它能够禁止该屏障之前的任何读操作与该屏障之后的任何读、写操作之间进行重排序。释放屏障相当于 `LoadStore` 屏障和 `StoreStore` 屏障的组合，它能够禁止该屏障之前的任何读、写操作与该屏障之后的任何写操作之间进行重排序。

11.5.1 volatile 关键字的实现

Java 虚拟机（JIT 编译器）在 `volatile` 变量写操作之前插入的释放屏障使得该屏障之前的任何读、写操作都先于这个 `volatile` 变量写操作被提交，而 Java 虚拟机（JIT 编译器）在 `volatile` 变量读操作之后插入的获取屏障使得这个 `volatile` 变量读操作先于该屏障之后的任何读、写操作被提交。写线程和读线程通过各自执行的释放屏障和获取屏障保障了有序性。

假设写线程、读线程依照表 11-7 中的线程交错顺序（即读线程执行时，写线程对共享变量的更新操作已经完成）执行，`A`、`B` 是普通共享变量，`V` 是 `volatile` 变量。释放屏障确保了写线程对共享变量 `A`、`B` 的更新会先于对 `V` 的更新被提交，这就意味着读线程在读取到写线程对 `V` 的更新情况下也能够读取到写线程对 `A` 和 `B` 的更新。为了保障读线程对写线程所执行的写操作的感知顺序与程序顺序一致，读线程必须依照与写线程的程序顺序的相反顺序即先读取 `V` 再读取 `A` 或者 `B` 来执行读操作。由于读线程中的读操作（或写操作）也可能被重排序（包括指令重排序和内存重排序），因此 Java 虚拟机会在读线程中的 `volatile` 读操作之后插入一个获取屏障，以保证该线程对变量 `V` 的读取操作先于对 `A`、`B` 的读取操作被提交。写线程、读线程通过释放屏障和获取屏障的这种配对使用保障了读线程对写线程执行的写操作的感知顺序与程序顺序一致，即保障了有序性。

表 11-7 volatile 有序性与内存屏障

写线程	读线程
<code>A=1;</code> <code>B=2;</code> <code>[LoadStore+StoreStore]//释放屏障</code> <code>V=true;</code>	

续表

写线程	读线程
	<pre>if(V){ [LoadLoad+LoadStore]//获取屏障 sum=A+B; }</pre>

需要注意的是，释放屏障只是确保了该屏障之前的读、写操作先于该屏障之后的任何写操作被提交，因此释放屏障之前的操作之间，其提交顺序可以与程序顺序不一致。例如，针对表 11-7 中写线程对 A、B 的更新，处理器并无须保证对 A 的更新先于对 B 的更新被提交，而只需要保障对 A 以及 B 的更新先于对 V 的更新被提交即可。类似地，获取屏障只是确保了该屏障之前的任何读操作先于该屏障之后的任何读、写操作被提交。因此获取屏障之后的操作之间，其提交顺序可以与程序顺序不一致。写线程和读线程通过配对使用释放屏障和获取屏障，使得上述内存操作提交顺序与程序顺序的不一致并不会对有序性产生影响。例如，就表 11-7 中的例子而言，写线程对 A 的更新（记为 W[A]）、对 B 的更新（记为 W[B]）以及对 V 的更新（记为 W[V]）在释放屏障的作用下呈现出的提交顺序可能是 W[A]→W[B]→W[V]，也可能是 W[B]→W[A]→W[V]，但无论如何处理器总是会保障 (W[A], W[B])→W[V]。而读线程在获取屏障的作用下总是先读取 W[V] 的结果，在此前提下该线程无论先读取 W[A] 的结果还是先读取 W[B] 的结果，都不会影响其读取到写线程对 A 和 B 的更新，因为 W[A]、W[B] 总是先于 W[V] 被提交。

Java 虚拟机（JIT 编译器）会在 volatile 变量写操作之后插入一个 StoreLoad 屏障。该屏障不仅禁止该屏障之后的任何读操作与该屏障之前的任何写操作（包括该 volatile 写操作）之间进行重排序，它还起到以下两个作用。

- 充当存储屏障。StoreLoad 屏障是一个通用存储屏障，其功能涵盖了其他 3 个基本内存屏障。StoreLoad 屏障通过清空其执行处理器的写缓冲器使得该屏障前的所有写操作（包括 volatile 写操作以及其他任何写操作）的结果得以到达高速缓存，从而使这些更新对其他处理器而言是可同步的。
- 充当加载屏障，以消除存储转发的副作用。假设处理器 Processor 0 在 t_1 时刻更新了某个 volatile 变量，在随后的 t_2 时刻又读取了该变量。由于存储转发技术可能使得一个处理器无法将其他处理器对共享变量所做的更新同步到该处理器的高速缓存上，而 Java 语言规范又要求 volatile 读操作总是可以读取到其他处理器对相应变量所做的更新，因此 Java 虚拟机需要在 volatile 变量写操作和随后的 volatile 变量读操作之间插入一个 StoreLoad 屏障。这是利用了 StoreLoad 屏障既能够清空写缓冲器还能够清空无效化队列的功能，从而使其他处理器对 volatile 变量所做的更新能够被同步到 volatile 变量读线程的执行处理器上。

Java 虚拟机 (JIT 编译器) 在 `volatile` 变量读操作前插入的一个加载屏障相当于 `LoadLoad` 屏障, 它通过清空无效化队列来使得其后的读操作 (包括 `volatile` 读操作) 有机会读取到其他处理器对共享变量所做的更新。读线程能够读取到写线程对 `volatile` 变量所做的更新, 有赖于写线程在 `volatile` 写操作后所执行的存储屏障。可见, `volatile` 对可见性的保障是通过写线程、读线程配对使用存储屏障和加载屏障实现的。

Java 虚拟机对 `synchronized` 关键字的实现方式与对 `volatile` 的实现方式类似。Java 虚拟机在 `monitorenter` (申请锁) 字节码指令对应的机器码指令之后临界区开始之前的地方所插入的获取屏障以及在 `monitorexit` (释放锁) 字节码指令对应的机器码指令之前临界区结束之后的地方所插入的释放屏障确保了临界区中任何读、写操作无法被重排序到临界区之外, 这一点再加上锁的排他性确保了临界区中的操作成为一个原子操作。

由于 x86 处理器仅支持 `StoreLoad` 重排序, 因此在 x86 处理器下 Java 虚拟机会将 `LoadLoad` 屏障、`LoadStore` 屏障以及 `StoreStore` 屏障映射为空指令。也就是说, x86 处理器下的 Java 虚拟机无须在 `volatile` 读操作前、`volatile` 读操作后以及 `volatile` 写操作前插入任何指令, 而只需要在 `volatile` 写操作后插入一个 `StoreLoad` 屏障, 这个屏障在 Hotspot 虚拟机中是由一个 `Lock` 前缀的空操作指令充当的¹³。而在其他处理器下, Java 虚拟机则可能根据相应处理器对重排序的支持情况在 `volatile` 读、写前后的相应地方插入相应的指令。

11.5.2 `synchronized` 关键字的实现

Java 虚拟机 (JIT 编译器) 会在 `monitorenter` (用于申请锁的字节码指令) 对应的指令后临界区开始前的地方插入一个获取屏障。Java 虚拟机会在临界区结束后 `monitorexit` (用于释放锁的字节码指令) 对应的指令前的地方插入一个释放屏障。这里, 获取屏障和释放屏障一起保障了临界区内的任何读、写操作都无法被重排序到临界区之外, 再加上锁的排他性, 这使得临界区内的操作具有原子性。

`synchronized` 关键字对有序性的保障与 `volatile` 关键字对有序性的保障实现原理是一样的, 也是通过释放屏障和获取屏障的配对使用实现的。释放屏障使得写线程在临界区中执行的读、写操作先于 `monitorexit` 对应的指令 (相当于写操作) 被提交, 而获取屏障使得读线程必须在获得锁 (相当于 `read-modify-write` 操作) 之后才能够执行临界区中的操作。写线程以及读线程通过这种释放屏障和获取屏障的配对使用实现了有序性。

¹³ 具体的指令是 “`lock addl $0x0, (%rsp)`”, `lock` 前缀指令能够清空写缓冲器, 而 x86 处理器并没有使用无效化队列, 因此该指令就起到了 `StoreLoad` 屏障的作用。

Java 虚拟机也会在 `monitorexit` 对应的指令（相当于写操作）之后插入一个 `StoreLoad` 屏障。这个处理的目的是与在 `volatile` 写操作之后插入一个 `StoreLoad` 屏障类似。该屏障充当了存储屏障，从而确保锁的持有线程在释放锁之前所执行的所有操作的结果能够到达高速缓存，并消除了存储转发的副作用。另外，该屏障禁止了 `monitorexit` 对应的指令与其他同步块的 `monitorenter` 对应的指令进行重排序，这保障了 `monitorenter` 与 `monitorexit` 总是成对的，从而使 `synchronized` 块的并列（一个 `synchronized` 块之后又有其他 `synchronized` 块）以及 `synchronized` 块的嵌套（一个 `synchronized` 块内包含其他 `synchronized` 块）成为可能。

11.5.3 Java 虚拟机对内存屏障使用的优化

内存屏障部分禁止重排序的代价就是它会阻止编译器（JIT 编译器）、处理器做一些性能优化。这就好比我们在日常生活中打乱预定的顺序往往可以提高办事效率，而一味地按照预定的顺序办事反而可能降低办事效率。比如，我们原先打算出门先去理发再去超市购物。结果，当我们到理发店的时候发现排队的顾客比较多，此时我们可以先去超市购物（因为购物也比较耗时）之后再去做理发（打乱预定的顺序），这样只要在我们购物期间理发店没有新来的顾客（或者新来的顾客比较少），那么我们在理发店等待的时间就会大为减少。

内存屏障的另外一种代价就是其实现往往涉及冲刷写缓冲器和清空无效化队列，而这两个动作可能是比较耗时的。

因此，Java 虚拟机对内存屏障的使用往往会做一些优化。这些优化包括省略、合并等。例如，对于两个连续的 `volatile` 写操作，Java 虚拟机可能只在最后一个 `volatile` 写操作之后插入 `StoreLoad` 屏障，而不是在每个 `volatile` 写操作后插入一个 `StoreLoad` 屏障。在 x86 处理器下，Java 虚拟机对 `monitorexit` 的实现本身就带有 `StoreLoad` 屏障的效果，因此 Java 虚拟机不会在 `monitorexit` 对应的机器码指令之后插入 `StoreLoad` 屏障。

11.5.4 final 关键字的实现

清单 3-28 中的 `HTTPRangeRequest` 类包含一个非 `final` 字段 `url` 和一个引用型 `final` 字段 `range`。如下语句：

```
sharedRef = new HTTPRangeRequest("http://xyz.com/download/big.tar",0,1048576);
```

在 JIT 编译器的内联（`Inline`）优化的作用下可能会被编译成与如下伪代码等效的指

令（伪代码表示）：

```
objRef = allocate(HTTPRangeRequest.class); // 子操作①：分配对象所需的存储空间
objRef.url = "http://xyz.com/download/big.tar"; // 子操作②：初始化普通字段
objRange = allocate(Range.class);
objRange.lowerBound = 0; // 子操作③：初始化 objRange
objRange.upperBound = 1048576; // 子操作④：初始化 objRange
objRef.range = objRange; // 子操作⑤：初始化 final 字段 range
sharedRef = objRef; // 子操作⑥：将对象引用写入共享变量 sharedRef
```

子操作①到子操作⑥之间的操作为 HTTPRangeRequest 类的构造器中的内容（指令）。Java 虚拟机会在子操作⑤（final 字段初始化）之后插入一个 StoreStore 屏障以禁止子操作⑤（final 字段初始化）以及该操作前的所有写操作和子操作⑥（对象发布）之间的重排序（包括指令重排序和内存重排序），从而使得 HTTPRangeRequest 实例引用 objRef 对外可见的时候，该实例的 final 字段以及这些 final 字段（引用型字段）所引用的对象已经初始化完毕。而 Java 虚拟机（JIT 编译器）在插入 StoreStore 屏障前可能将非 final 字段的初始化操作（子操作②）重排序（指令重排序）到子操作⑥之后，因此包含 final 字段的对象引用对外可见的时候该对象的非 final 字段仍然可能是未初始化完毕的。

由于某些处理器（比如 x86 处理器）可能不支持 StoreStore 重排序，因此运行在这种处理器上的 Java 虚拟机只需要保障其 JIT 编译器不将 final 字段的初始化操作重排序（指令重排序）到其构造器结束之后（即构造器之外，相当于上述的子操作⑥之后），而无需插入相应的 StoreStore 屏障。从性能的角度来说，在这些处理器平台上 final 关键字的开销并不大：相对于非 final 字段而言，final 字段的开销在于它阻止了 JIT 编译器可能做的一些优化（指令重排序）。

11.6 Java 内存模型

缓存一致性协议确保了一个处理器对某个内存地址进行的写操作的结果最终能够被其他处理器所读取。所谓“最终”就是带有不确定性，换言之，即一个处理器对共享变量所做的更新具体在什么时候能够被其他处理器读取这一点，缓存一致性协议本身是不保证的。写缓冲器、无效化队列都可能导致一个处理器在某一时刻读取到共享变量的旧值。因此，从底层的角度来看，计算机系统必须解决这样一个问题——一个处理器对共享变量所做的更新在什么时候或者说什么情况下才能够被其他处理器所读取，即可见性问题。可见性问题又衍生出新的问题——一个处理器先后更新多个共享变量的情况下，其他处理器是以何种顺序读取到这些更新的，即有序性问题。

用于回答上述问题的模型就被称为内存一致性模型（Memory Consistency Model），也被称为内存模型（Memory Model）。不同的处理器架构有着不同的内存模型，因此这些处理器对有序性的保障程度各异，这表现为它们所支持的内存重排序不同。例如，x86 处理器不支持 StoreStore 重排序（x86 仅支持 StoreLoad 重排序），这就意味着写线程所执行多个写操作在读线程看来其感知顺序与程序顺序一致。而 ARM 处理器则支持 4 种全部可能的重排序，因此一个处理器对另外一个处理器所执行的两个写操作的感知顺序可能与该处理器的程序顺序不一致。

Java 作为一个跨平台（跨操作系统和硬件）的语言，为了屏蔽不同处理器的内存模型差异，以便 Java 应用开发人员不必根据不同的处理器编写不同的代码，它必须定义自己的内存模型，这个模型就被称为 Java 内存模型¹⁴。

由于并发性 Bug（Concurrency Bug）往往不是在测试过程中而是在程序运行在高负荷（Heavy Load）的情况下才显现出来的，并且这种 Bug 难于再现，因此，第 10 章我们认为代码复审（Code Review）是挖掘并发性 Bug 的最有效途径。而了解和熟悉 Java 内存模型有助于我们在无须真正执行代码的情况下对多线程代码的执行结果进行推断（Reason），而这正是对多线程代码进行代码复审所依赖的基础。因此，有关 Java 内存模型的知识有助于我们做好代码复审。

11.6.1 什么是 Java 内存模型

Java 内存模型（Java Memory Model, JMM）是 Java 语言规范（The Java Language Specification, JLS）的一部分，它最初主要是在 JLS 的第 17 章描述的。Java 内存模型定义了 final、volatile 和 synchronized 关键字的行为并确保正确同步（Correctly Synchronized）的 Java 程序能够正确地运行在不同架构的处理器之上。从应用开发人员的角度来看，Java 内存模型作为一个模型，它从“什么”（What）的角度为我们解答以下几个线程安全方面的问题。

- 原子性问题。针对实例变量、静态变量（即共享变量而非局部变量）的读、写操作，哪些是具备原子性的，哪些可能不具备原子性？
- 可见性问题。一个线程对实例变量、静态变量（即共享变量）进行的更新在什么情况下能够被其他线程所读取？
- 有序性问题。一个线程对多个实例变量、静态变量（即共享变量）进行的更新在什么情况下在其他线程看来可以是乱序的（即感知顺序与程序顺序不同）。

14 如果需要那样做的话，那么 Java 岂不成了一门汇编语言？

在原子性方面, Java 内存模型规定对 long/double 型以外的基本数据类型以及引用类型的共享变量进行读、写操作都具有原子性。另外, Java 内存模型还特别规定对 volatile 修饰的 long/double 型共享变量进行读、写操作也具有原子性。换言之, 对引用类型以及几乎所有基本数据类型的共享变量进行的读、写操作, Java 内存模型都保证它们具有原子性, 而对 long/double 型的共享变量进行的读、写操作是否具有原子性则取决于具体的 Java 虚拟机实现。

对于可见性问题和有序性问题, Java 内存模型则使用 happens-before 这个术语来解答。

11.6.2 happen(s)-before 关系

在介绍 happens-before 之前, 我们换个角度来理解有序性这个概念——用可见性描述有序性。假设处理器 Processor 0、Processor 1 上的两个线程依照表 11-8 所示的线程交错顺序执行, X、Y、Z 和 ready 为共享变量, r1、r2 和 r3 为局部变量。进一步假设, Processor 1 读取到变量 ready 值时 S1、S2、S3 和 S4 的操作结果均已提交完毕, 并且 L2、L3 不会与 L1 进行重排序¹⁵, 那么此时 S1、S2、S3 和 S4 的操作结果对 L1 及其之后(程序顺序)的 L2 和 L3 来说都是可见的。因此, 从 L1、L2 和 L3 的角度来看此时 S1、S2、S3 和 S4 就像是 Processor 0 上的线程依照程序顺序执行一样, 即 S1、S2、S3 和 S4 对于 L1、L2 和 L3 来说是有序的。尽管实际上 Processor 0 在执行 S1、S2、S3 和 S4 时可能进行指令重排序、内存重排序, 但是只要在 L1 被执行的时候 S1、S2、S3 和 S4 的操作结果均已提交完毕, 即这些操作的结果同时对 L1 可见, 那么 S1、S2、S3 和 S4 在 L1、L2 和 L3 看来就是有序的。

表 11-8 用可见性来阐述有序性示例

Processor 0	Processor 1
X=1;//S1	
Y=2;//S2	
Z=3;//S3	
ready=true;//S4	
	r1=ready;//L1
	r2=X+Y;//L2
	r3=Z;//L3

¹⁵ 例如, ready 是个 volatile 变量就可以满足这个假设。

happens-before 就是采用上述这种从可见性角度出发去描述有序性的。Java 内存模型定义了一些动作（Action）。这些动作包括变量的读/写、锁的申请（lock）与释放（unlock）以及线程的启动（Thread.start()调用）和加入（Thread.join()调用）等。假设动作 A 和动作 B 之间存在 happens-before 关系（happens-before relationship），称之为 A happens-before B，那么 Java 内存模型保证 A 的操作结果对 B 可见，即 A 的操作结果会在 B 被执行前提交（比如写入高速缓存或者主内存）。happens-before 关系具有传递性（Transitivity），即如果 A happens-before B，并且 B happens-before C，那么有 A happens-before C。

约定

为方便表述，后续我们将使用“ \rightarrow ”这个符号来表示 happens-before 关系，比如 $A \rightarrow B$ 表示动作 A happens-before 动作 B。

happens-before 关系中的两个动作既可以是同一个线程执行的，也可以是不同线程执行的。上述例子中的可见性和有序性可以用 happens-before 关系来描述。我们不妨假设 S1、S2 和 S3 中的任意一个操作与 S4 都具有 happens-before 关系，记为 $(S1, S2, S3) \rightarrow S4$ ；L1 与 L2、L3 中的任意一个操作都具有 happens-before 关系，记为 $L1 \rightarrow (L2, L3)$ ；并且 S4 happens-before L1，记为 $S4 \rightarrow L1$ ¹⁶；那么由于 happens-before 关系具有传递性，可有 $(S1, S2, S3, S4) \rightarrow (L1, L2, L3)$ ，即 S1、S2、S3 和 S4 中的任意一个操作与 L1、L2 和 L3 中的任意一个操作都具有 happens-before 关系，因此 S1、S2、S3 和 S4 的操作结果对 L1、L2 和 L3 中的任意一个操作可见。这意味着 S1、S2、S3 和 S4 在 L1、L2 和 L3 中的任意一个操作看来像是按照程序顺序提交的，即这些操作在 L1、L2 和 L3 看来是有序的。

happens-before 关系的传递性使得可见性保障具有累积的效果。假设 A、B、C 和 D 四个动作之间存在这样的 happens-before 关系： $A \rightarrow B$ 、 $B \rightarrow C$ 、 $C \rightarrow D$ 。根据传递性，我们还可以推导出这样的 happens-before 关系： $A \rightarrow D$ 和 $B \rightarrow D$ （当然，还有 $A \rightarrow C$ ）。根据 happens-before 关系本身蕴含的可见性保障结合上述 happens-before 关系（包括假设中存在的以及推导出来的 happens-before 关系）可知：对于 D 而言，不仅仅 C 的结果对其可见，A 的结果以及 B 的结果也都对 D 可见，这就形成了可见性保障能够“累积”的效果，如图 11-8 所示（图中虚线箭头表示推导出来的 happens-before 关系）。

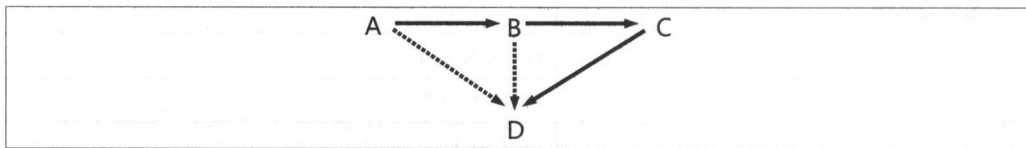


图 11-8 happens-before 关系与可见性保障的累积

16 下文会介绍这种假设成立的理由。

如果一组动作 ($\{A_1, A_2, A_3\}$) 中的每个动作与另外一组动作 ($\{B_1, B_2, B_3\}$) 中的任意一个动作都具有 happens-before 关系, 那么我们可以称前一组动作与后一组动作之间存在 happens-before 关系, 记为 $\{A_1, A_2, A_3\} \rightarrow \{B_1, B_2, B_3\}$ 。因此, happens-before 关系与 happens-before 关系本质上是一回事: 从数学上的集合角度来看, happens-before 关系可以理解两个集合 (两组动作) 之间的关系, 而 happens-before 关系也可以被看作两个集合之间的关系, 只不过这两个集合各自都只包含一个元素。

Java 内存模型定义了一些关于 happens-before 关系的规则, 这些规则规定了两个动作在什么情况下具有 happens-before 关系。其中常用的规则如下。

- 程序顺序规则 (Program Order Rule)。该规则即前面章节所说的“貌似串行语义” (As-if-serial Semantics)。一个线程中的每一个动作都 happens-before 该线程中程序顺序上排在该动作之后的每一个动作。

程序顺序规则意味着一个线程内任何一个动作的结果对程序顺序上该动作之后的其他动作都是可见的, 并且这些动作在该线程自身看来就像是完全依照程序顺序执行和提交的。尽管如此, 只要这些动作之间不存在数据依赖关系, 那么 Java 虚拟机 (JIT 编译器)、处理器都可能对这些动作进行重排序, 只要这种重排序不违反程序顺序规则即可。因此, 程序顺序上先后的两个动作 A 和 B, 尽管它们之间存在 happens-before 关系, 但这并不意味着在时间上动作 A 必须先于动作 B 被执行。由此可见, happens-before 关系与时间上的先后关系并无必然的联系。

- 内部锁规则 (Monitor Lock Rule)。内部锁的释放 (unlock) happens-before 后续 (Subsequent) 每一个对该锁的申请 (lock)。理解这个规则有两点需要注意: 首先, 该规则中的“释放”和“申请”必须是针对同一锁实例, 也就是说一个锁的释放与另外一个锁的申请之间并无 happens-before 关系; 其次, 所谓“后续”是指时间上的先后关系, 即一个线程释放锁后另外一个线程再来申请这个锁的情况下, 这两个线程的“释放”和“申请”之间才存在 happens-before 关系。这就是我们在介绍线程同步机制的时候强调访问同一组共享变量的线程必须同步在同一个锁实例之上的原因——不是同步在同一个锁实例之上就无法保证 happens-before 关系, 因此也就无法保证可见性和有序性。

内部锁规则和程序顺序规则一起确保了锁对可见性和有序性的保障。设线程 T_1 、 T_2 是同步在锁 M 之上的两个线程, T_2 在 T_1 释放 M 之后申请了 M, 如图 11-9 所示。根据程序顺序规则可有 happens-before 关系 hb_1 和 happens-before 关系 hb_3 , 根据内部锁规则可有 happens-before 关系 hb_2 , 结合 happens-before 关系的传递性可有 happens-before 关系 hb_4 。由 hb_4 可知, T_1 在释放锁前所执行的任何动作的结

果对 T_2 在获得锁（申请锁成功）之后所执行的任何一个动作可见。因此，从 T_2 在获得锁之后所执行的动作看来， T_1 在释放锁前所执行的一系列动作就像是完全依照程序顺序执行和提交的，即这些动作是有序的。由 happens-before 关系导致的可见性的累积效果可知， T_1 在 unlock 前所执行的任何动作的结果对 T_2 在 lock 之后所执行的任何动作而言都是可见的。也就是说， T_1 在临界区之中以及临界区之前所执行的所有动作的结果对 T_2 在 lock 之后所执行的任何动作而言都是可见的。只不过 T_2 读取共享变量的时候，如果这些共享变量是由 T_1 在临界区前所更新的，那么 Java 语言并不保证 T_2 读取到的值是最新的（因为 T_2 读取这些变量的时候，可能有其他线程正在执行临界区前的代码）。可见，尽管锁对排他性的保障仅限于临界区内的代码，但是锁对可见性和有序性的保障却可以扩展到临界区之前。

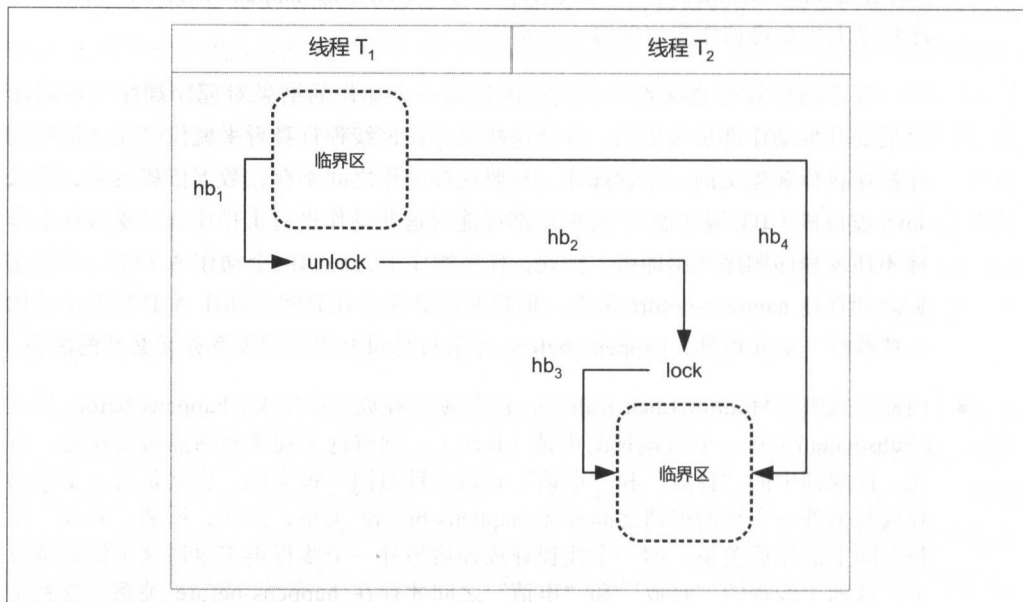


图 11-9 happens-before 关系与锁对可见性、有序性的保障

提示

锁对排他性的保障仅限于临界区内的代码，但是锁对可见性和有序性的保障却可以扩展到临界区之前。

- volatile 变量规则（Volatile Variable Rule）。对一个 volatile 变量的写操作 happens-before 后续（Subsequent）每一个针对该变量的读操作。理解这个规则有两点需要注意：首先，针对同一个 volatile 变量的写、读操作之间才有 happens-before 关系，不同 volatile 变量之间的写、读操作并无 happens-before 关

系；其次，针对同一个 `volatile` 变量的写、读操作必须具有时间上的先后关系，即一个线程先写，另外一个线程再来读才能够有 `happens-before` 关系。

`volatile` 关键字能够对可见性和有序性进行保障这一点同样也可以通过 `happens-before` 关系推导出来，推导过程与上述对锁对可见性和有序性进行保障类似。

表 11-8 的例子所做的假设之所以能够成立，就是因为程序顺序规则、`volatile` 变量规则（或者内部锁规则）的存在。

- 线程启动规则（Thread Start Rule）。调用一个线程的 `start` 方法 `happens-before` 被启动的这个线程中的任何一个动作。假设线程 T_1 在执行过程中启动了线程 T_2 ，即 T_1 执行了 `T_2.start()`，那么线程启动规则会保证 T_1 在 `T_2.start()` 调用前所执行的任何动作的结果对 T_2 所执行的任何一个动作都是可见的，并因此是有序的。
- 线程终止规则（Thread Termination Rule）。一个线程中的任何一个动作都 `happens-before` 该线程的 `join` 方法的执行线程在 `join` 方法返回之后所执行的任意一个动作。假设线程 T_1 等待线程 T_2 结束，那么线程终止规则保证 T_2 所执行的任何动作的结果对 T_1 中程序顺序上在 `T_2.join()` 调用之后的任何一个动作是可见的，并因此是有序的。

扩展阅读 Java 内存模型定义的基本 `happens-before` 规则（英文）

-
- **Program Order Rule.** Each action in a thread happens-before every action in that thread that comes later in the program's order.
 - **Monitor Lock Rule.** An unlock (synchronized block or method exit) of a monitor happens-before every *subsequent* lock (synchronized block or method entry) of *that same* monitor.
 - **Volatile Variable Rule.** A write to a volatile field happens-before every *subsequent* read of *that same* field.
 - **Thread Start Rule.** A call to start on a thread happens-before any action in the started thread.
 - **Thread Termination Rule.** All actions in a thread happen-before any other thread successfully returns from a join on that thread.
-

Java 标准库类也定义了一些 happens-before 规则（关系），这些规则建立在 Java 内存模型所定义的基本 happens-before 规则之上¹⁷。例如，对于任意的 `CountDownLatch` 实例 `countDownLatch`，一个线程在 `countDownLatch.countDown()`调用前所执行的所有动作与另外一个线程在 `countDownLatch.await()`调用成功返回之后所执行的所有动作之间存在 happen-before 关系；对于任意的 `BlockingQueue` 实例 `blockingQueue`，一个线程在 `blockingQueue.put(E)`调用所执行的所有动作与另外一个线程在 `blockingQueue.take()`调用返回之后所执行的所有动作之间存在 happen-before 关系。

我们知道从应用代码的层次来看，可见性和有序性的保障是通过应用代码使用 Java 线程同步机制实现的。换言之，无论是 Java 内存模型定义的 happens-before 规则，还是 Java 标准库类定义的 happens-before 规则，从应用程序层面来看，它们都是通过使用 Java 线程同步机制实现的。Java 内存模型定义的基本 happens-before 规则除程序顺序规则以外，其他规则均涉及 Java 同步机制：内部锁规则涉及内部锁或者显式锁；`volatile` 变量规则涉及 `volatile` 关键字；线程启动规则和线程终止规则所涉及的 `Thread.start()/Thread.join()`的内部实现都依赖于锁（内部锁），因此这两个规则实际上也是依赖于锁。而 Java 标准库类定义的 happens-before 规则其实是可以通过基本 happens-before 规则推导出来的，因此这些规则的实现实际上也是应用 Java 线程同步机制实现的结果。Java 内存模型将一个程序中的所有动作看作一个集合。该集合中的任意两个动作之间可能存在 happens-before 关系，也可能不存在 happens-before 关系。只有正确地使用同步机制的两个动作之间才存在 happens-before 关系，从而使可见性、有序性有所保障。相反，未使用（或者未正确使用）同步机制的两个动作之间由于缺乏 happens-before 关系而不具有可见性、有序性保障。换言之，这两个动作之间的可见性、有序性无法得到 Java 虚拟机本身的保障，而是取决于具体的处理器本身。

Java 内存模型作为一个模型，它只会从“什么”（What）而不会从“如何”（How）的角度来描述 Java 语言对可见性、有序性的保障。这里的“什么”便是上述的 happen(s)-before 关系。这些 happens-before 规则最终是由 Java 虚拟机、编译器以及处理器一同协作来落实的，而内存屏障则是 Java 虚拟机、编译器和处理器之间的“沟通”纽带。例如，为了实现程序顺序规则和内部锁规则，我们必须确保一个线程在执行 `unlock (monitorexit)`前所执行的操作不能够被重排序（包括指令重排序、内存重排序）到 `unlock` 之后，即必须确保 `unlock` 前所执行的操作先于 `unlock` 本身被提交，否则我们无法保证程序顺序上在 `unlock` 之前的动作结果对后续执行 `lock` 的线程可见。我们知道，Java 虚拟机、编译器以及处理器都可能进行重排序。因为 Java 代码的编译与执行是由 Java 编译器（主要是 JIT 编译器）

17 参见：<http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/package-summary.html#MemoryVisibility>。

与 Java 虚拟机落实的，所以 Java 虚拟机以及 Java 的 JIT 编译器本身就不会将程序顺序上在 unlock 前的操作重排序到 unlock 之后¹⁸，而处理器所执行的代码是机器语言代码（而不是 Java 这种高级语言代码）。因此，为了确保处理器也遵守这些 happens-before 规则，Java 虚拟机、Java 的 JIT 编译器会在 monitorexit 对应的机器码指令前插入一个释放屏障，这相当于告诉处理器禁止相应的重排序¹⁹。

11.6.3 再探对象的安全发布

了解 happen(s)-before 规则之后，我们便可以从 Java 内存模型的角度来重新审视 volatile 关键字和锁对可见性和有序性的保障以及对象的安全发布。

根据 Java 内存模型以及 Java 标准库类定义的 happen(s)-before 规则，我们便可以知道任意两个动作之间是否存在 happens-before 关系。由此，我们便可以推断出两个线程所执行的操作之间的可见性和有序性是否有保障。例如，在第 4 章的第 2 个实战案例（响应延时统计程序）中，有个 RecordSet 类（日志记录集，完整代码参见清单 4-8），如下代码片段所示：

```
public class RecordSet {
    public final int capacity;
    final String[] records;
    int readIndex = 0;
    int writeIndex = 0;

    public RecordSet(int capacity) {
        this.capacity = capacity;
        records = new String[capacity];
    }

    public String nextRecord() {
        String record = null;
        if (readIndex < writeIndex) {
            record = records[readIndex++];
        }
        return record;
    }
}
```

18 Java 编译器可能将临界区前的操作重排序到临界区内，但是不会将其重排序到 monitorexit 对应的机器码指令之后。

19 对有的 Java 虚拟机（比如 Hotspot 虚拟机）而言，JIT 编译器是其一个模块。而对有些 Java 虚拟机来说，JIT 编译器是另外一个与之相对独立的软件。因此，这里的 Java 虚拟机从其作用上可以理解为 JIT 编译器。

```

public boolean putRecord(String line) {
    if (writeIndex == capacity) {
        return true;
    }
    records[writeIndex++] = line;
    return false;
}
// ...
}

```

约定

为便于讲解，下文将 11.6.2 节介绍的由 Java 标准库中定义的有关 `BlockingQueue` 的 `happen-before` 关系称为 `BlockingQueue` 规则，亦即：对于 `BlockingQueue` 实例 `q`，`q.put(E)` 的执行线程在该调用前所执行的所有动作，与 `q.take()` 方法的执行线程在该方法调用返回之后所执行的所有动作之间存在 `happen-before` 关系。

`RecordSet` 实例会被日志文件读取线程（参见清单 4-10）和统计处理线程（参见清单 4-7）共享。其中，日志文件读取线程会执行 `RecordSet.putRecord(String)`，这涉及实例变量 `writeIndex` 的更新；而统计处理线程则会执行 `RecordSet.nextRecord()`，这涉及实例变量 `writeIndex` 的读取。尽管 `writeIndex` 是这两个线程之间的共享变量，但是我们并没有在 `putRecord/nextRecord` 方法中以及 `writeIndex` 本身的声明中使用任何线程同步机制。然而，这并不会导致线程安全（可见性）问题。这是因为日志文件读取线程在调用完 `RecordSet.putRecord(String)` 之后会将相应的 `RecordSet` 实例存入（`BlockingQueue.put(E)` 调用）一个 `BlockingQueue`，而统计处理线程则从该 `BlockingQueue` 中取出（`BlockingQueue.take()` 调用）相应的 `RecordSet` 实例后才调用这个实例的 `nextRecord()` 方法。根据上述有关 `BlockingQueue` 的 `happen-before` 规则（`BlockingQueue` 规则），日志文件读取线程对 `writeIndex` 的更新操作（它是在 `BlockingQueue.put(E)` 调用前执行的动作）与统计处理线程对 `writeIndex` 的读取操作（它是在 `BlockingQueue.take()` 调用后执行的动作）之间存在 `happens-before` 关系，因此日志文件读取线程对 `writeIndex` 所做的更新对统计处理线程是可见的。

这个例子从对象发布的角度来看，我们可以说日志文件读取线程借助 `BlockingQueue` 将 `RecordSet` 实例以线程安全的方式发布到统计处理线程。这个发布不仅仅使统计处理线程“看到”一个 `RecordSet` 实例的引用，它还使日志文件读取线程对 `RecordSet` 实例所做的更新对统计处理线程来说是可见的且有序的（下面会解释），尽管 `RecordSet` 类本身几乎没有使用任何线程同步机制（除了使用 `final` 修饰实例变量 `capacity`）。

实际上, `BlockingQueue` 规则可以通过程序顺序规则结合 `BlockingQueue` 内部使用的显式锁所建立起的内部锁规则推导出来。日志文件读取线程执行 `BlockingQueue.put(E)` 相当于释放一个锁 `M`, 而后续统计处理线程执行 `BlockingQueue.take()` 则相当于申请锁 `M`, 那么根据内部锁规则可有 `BlockingQueue.put(E)` 调用 → 后续的 `BlockingQueue.take()` 调用²⁰。再结合程序顺序规则与 `happens-before` 关系的传递性, 我们不难推断出日志文件读取线程在 `BlockingQueue.put(E)` 调用前所执行的所有动作 (包括更新 `writeIndex`) → 统计处理线程在 `BlockingQueue.take()` 调用后所执行的所有动作 (包括读取 `writeIndex`)。也就是说, 日志文件读取线程在发布一个 `RecordSet` 实例前所执行的所有动作, 与统计处理线程从队列中取出该 `RecordSet` 实例后所执行的所有动作之间存在 `happen-before` 关系。显然, 日志文件读取线程在 `BlockingQueue.put(E)` 调用前所执行的所有动作并不位于 `BlockingQueue` 内部所使用的锁的临界区之内, 然而这些动作对统计处理线程在 `BlockingQueue.take()` 调用后所执行的所有动作都是可见的且有序的。由此可见, 程序顺序规则与 `happens-before` 关系的传递性, 使得锁对可见性和有序性的保障从临界区内扩展到临界区之前。这才是对象安全发布的实质——不仅仅使一个对象的引用对其他线程可见, 还要保障该对象的引用对其他线程可见前, 发布线程对该对象所执行的操作对其他线程来说是可见的且有序的 (尽管这些操作并没有在临界区中执行)。

由于 `volatile` 变量规则与内部锁规则相似, 因此借助一个 `volatile` 变量也能够实现对象的安全发布。例如, 第 3 章我们在介绍基于双重检查锁定法的单例类的正确实现时强调静态变量 `instance` 必须使用 `volatile` 关键字修饰, 如下代码所示:

```
public class DCLSingleton {
    /*
     * 保存该类的唯一实例, 使用 volatile 关键字修饰 instance
     */
    private static volatile DCLSingleton instance;

    /*
     * 私有构造器使其他类无法直接通过 new 创建该类的实例
     */
    private DCLSingleton() {
        // 什么也不做
    }

    /**
     * 创建并返回该类的唯一实例 <BR>
```

20 实际上, `LinkedBlockingQueue` 内部使用的锁有两个。因此, 这里所说的锁 `M` 是从逻辑层面而非物理层面来说的。


```

    * 即只有该方法被调用时该类的唯一实例才会被创建
    *
    * @return
    */
    public static DCLSingleton getInstance() {
        if (null == instance) { // 操作①：第 1 次检查
            synchronized (DCLSingleton.class) {
                if (null == instance) { // 操作②：第 2 次检查
                    instance = new DCLSingleton(); // 操作③
                }
            }
        }
        return instance;
    }

    public void someService() {
        // 省略其他代码
    }
}

```

这里，假设线程 T_1 刚刚执行完操作③并退出临界区的时候，线程 T_2 恰好执行到操作①，尽管对 `instance` 的赋值以及对应对象的初始化是在临界区中执行的，但是 T_2 此时读取 `instance` 变量（操作①）并不是在临界区中进行的，故而 T_1 和 T_2 这时执行的两个动作之间并无 happens-before 关系，因此 T_1 所执行的动作对 T_2 来说可见性和有序性均无法得以保障。相反，如果我们用 `volatile` 修饰 `instance`，那么 T_1 对 `instance` 进行的操作与 T_2 对 `instance` 的读取操作之间借助于 `volatile` 所建立的 happen-before 关系（`volatile` 变量规则）便有了 happen-before 关系。这就相当于 T_1 将 `instance` 所引用的对象安全地发布到 T_2 ，从而使 T_2 一旦读取到 `instance` 值不为 `null`，那么该变量所引用的对象必然已经是初始化完毕的（即构造器中的操作都已经执行结束，保障这一点才能使延迟加载的单例模式得以正确实现）。

提示

- 程序顺序规则与 happens-before 关系的传递性，使得 `volatile` 关键字/锁对可见性和有序性的保障从临界区内扩展到临界区之前。
- 对象的安全发布不仅仅意味着使一个对象的引用对其他线程可见，它还意味着我们要保障该对象的引用对其他线程可见前，发布线程对该对象所执行的操作（即使这些操作并没有在临界区中执行）对其他线程来说是可见的且有序的。

11.6.4 JSR 133

早期（JDK 1.5 之前）的 Java 内存模型（在 Java 语言规范的第 17 章中定义）陆续被发现存在若干严重的缺陷，这些缺陷会导致多线程程序出现一些令人困惑的行为并阻碍了编译器执行一些常见的优化²¹。例如，早期的 Java 内存模型可能使一个线程先看到一个 final 字段的默认值接着才看到该字段的初始值，即 final 字段的值实际上可能是会变化的，这显然有悖 final 关键字的语义；早期的 Java 内存模型允许 volatile 写操作与其他的非 volatile 读、写操作进行重排序，这与多数开发人员对 volatile 关键字的直观感受不一致从而导致一些困惑。

为了修复早期的 Java 内存模型中存在的缺陷，JSR 133（133 号 Java Specification Request，又被称为 Java Memory Model）为 Java 语言定义了一个新的内存模型²²。

11.7 共享变量与性能

在多个线程共享变量的情况下，一个处理器对该变量进行更新会导致其他处理器上的高速缓存中存储的该变量的副本数据失效。这使得这些处理器后续访问（包括读、写）该变量时会产生缓存未命中，从而不利于程序的性能。多个线程之间对共享变量的访问仅仅涉及读操作而没有涉及写操作，或者这些线程之间不存在共享变量均有利于减少缓存未命中的频率。

11.8 本章小结

本章介绍了多线程编程的硬件基础以及 Java 内存模型的基础知识。本章知识结构如图 11-10 所示。

高速缓存是一个存取速率远比主内存大而容量远比主内存小的存储部件，其引入弥补了处理器与主内存处理能力之间的鸿沟。高速缓存相当于一个由硬件实现的散列表，其键为内存地址，其值为从内存读取或者准备写入内存的数据。高速缓存中的每个桶可包含若干缓存条目。缓存条目中的 Tag 部分包含了内存地址的高位部分比特；Flag 部分指示了缓存条目的有效性；缓存行用于存储从内存读取或者准备写入内存的数据，其容量在 16 ~ 256 字节之间不等，一个缓存行可用于存储多个变量。缓存命中意味着待读取或者写入内

21 参见：<https://www.ibm.com/developerworks/library/j-jtp02244/>。

22 详情参见：<https://www.cs.umd.edu/~pugh/java/memoryModel/jsr-133-faq.html>。

存的数据在高速缓存中存在相应的副本，这可以提升内存访问效率。缓存未命中包括读未命中和写未命中，它不利于性能，但是由于高速缓存容量的限制又往往是不可避免的。Linux 内核工具 perf 可用来查看缓存未命中情况。现代处理器多采用多级高速缓存，典型的高速缓存层级包括 L1 Cache、L2 Cache 和 L3 Cache。

缓存一致性协议保障了多个处理器上高速缓存中的数据副本的数据一致性，避免了一个处理器读取到共享变量的旧值以及避免了一个处理器对共享变量所做的更新丢失。MESI 协议是一个广为使用的缓存一致性协议，在该协议下缓存条目的 Flag 可能值包括：M/E/S/I。内存读/写操作是通过处理器发送与接收相关消息并更新缓存条目的 Flag 实现的。这些消息包括：Read/Read Response、Invalidate/Invalidate Acknowledge、Read Invalidate、Writeback。

写缓冲器与无效化队列的引入弥补了 MESI 协议的性能弱点。

写缓冲器是处理器内部的一个容量比高速缓存还小的私有高速存储部件。其引入使得内存写操作的执行处理器无须等待其他处理器回复 Invalidate Acknowledge/Read Response 消息便可以执行其他指令，从而减小内存写操作的延迟。写缓冲器能导致写线程对共享变量所做的更新无法被其他处理器同步过去。存储转发技术使得一个处理器可以直接从写缓冲器中读取该处理器先前执行的写操作的结果，但是它也可能导致可见性问题。另外，写缓冲器还会导致 StoreLoad 重排序和 StoreStore 重排序。

无效化队列的引入使得处理器在接收到 Invalidate 消息之后可以立即回复 Invalidate Acknowledge 消息，这减少了发送 Invalidate 消息的处理器等待时间。无效化队列可能使写线程对共享变量所做的共享无法反映到读线程执行处理器的高速缓存中，即导致可见性问题。无效化队列可以导致 LoadLoad 重排序。

从硬件的角度来看，可见性的保障是通过写线程和读线程配对使用存储屏障和加载屏障实现的。存储屏障能够冲刷写缓冲器使得写线程对共享变量所做的更新能够被其他处理器同步，加载屏障能够清空无效化队列，使得写线程对共享变量所做的更新能够反映在读线程执行处理器的高速缓存之中。

获取屏障相当于 LoadLoad 屏障和 LoadStore 屏障的组合，释放屏障相当于 StoreStore 屏障和 StoreLoad 屏障的组合。LoadLoad 屏障相当于加载屏障；而 StoreLoad 屏障是“全能型”屏障，它既可以充当存储屏障，也可以充当加载屏障。

Java 虚拟机（JIT 编译器）为了确保 final 关键字的语义，会在 final 字段初始化与构造器返回之前插入一个 StoreStore 屏障，这使得 final 字段初始化操作无法被重排序到构造器之外，从而确保了构造器返回之后相应对象的 final 字段总是初始化完毕的。有序性的保障是通过写线程与读线程配对执行释放屏障和获取屏障实现的，同样这些屏障也是 Java 虚拟机（JIT 编译器）替我们的应用程序插入的。Java 虚拟机（JIT 编译器）会在 volatile 变量写操作之后插入一个 StoreLoad 屏障，该屏障不仅充当了存储屏障以冲刷写缓冲器，它还充当了加载屏障以清空无效化队列从而消除了存储转发技术的副作用。Java 虚拟机（JIT 编译器）会在 volatile 变量读操作前插入一个 LoadLoad 屏障，该屏障充当了加载屏障，用于清空无效化队列。

Java 内存模型从“什么”（What）的角度来回答线程安全有关问题，JSR 133 对 Java 内存模型进行了增强和修复。Java 内存模型规定，long/double 型变量以外的任何变量的读/写操作具有原子性；volatile 变量修饰的 long/double 型变量的读/写操作也具有原子性。long/double 型普通变量的读/写操作的原子性取决于具体的 Java 虚拟机。happens-before 从可见性的角度对有序性进行描述。happens-before 关系具有传递性和累积效果。Java 内存模型定义的 happens-before 规则包括：程序顺序规则、内部锁规则、volatile 变量规则、线程启动规则和线程终止规则。Java 标准库本身也定义了一些 happens-before 规则。从语言的层面来看，这些规则是通过使用 Java 的同步机制实现的；从底层的角度来看，这些规则是由 Java 虚拟机、编译器以及处理器一同协作来落实的，内存屏障则是 Java 虚拟机、编译器和处理器之间的“沟通”纽带。



图 11-10 本章知识结构图

Java 多线程程序的性能调校

本章结合实战案例介绍与多线程程序紧密相关的常用性能调校方法与技术。

12.1 Java 虚拟机对内部锁的优化

自 Java 6/Java 7 开始, Java 虚拟机对内部锁的实现进行了一些优化。这些优化主要包括锁消除 (Lock Elision)、锁粗化 (Lock Coarsening)、偏向锁 (Biased Locking) 以及适应性锁 (Adaptive Locking)。这些优化仅在 Java 虚拟机 server 模式下起作用¹。

12.1.1 锁消除

锁消除 (Lock Elision) 是 JIT 编译器对内部锁的具体实现所做的一种优化, 如图 12-1 所示²。在动态编译同步块的时候, JIT 编译器可以借助一种被称为逃逸分析 (Escape Analysis) 的技术来判断同步块所使用的锁对象是否只能够被一个线程访问而没有被发布到其他线程。如果同步块所使用的锁对象通过这种分析被证实只能够被一个线程访问, 那么 JIT 编译器在编译这个同步块的时候并不生成 `synchronized` 所表示的锁的申请与释放对应的机器码, 而仅生成原临界区代码对应的机器码, 这就造成了被动态编译的字节码就像是不包含 `monitorenter` (申请锁) 和 `monitorexit` (释放锁) 这两个字节码指令一样, 即消除了锁的使用。这种编译器优化就被称为锁消除 (Lock Elision), 它使得特定情况下我们可以完全消除锁的开销。

1 即运行 Java 程序时我们可能需要在命令行中指定 Java 虚拟机参数 “-server” 以开启这些优化。

2 IBM J9 Java 虚拟机也支持该优化。

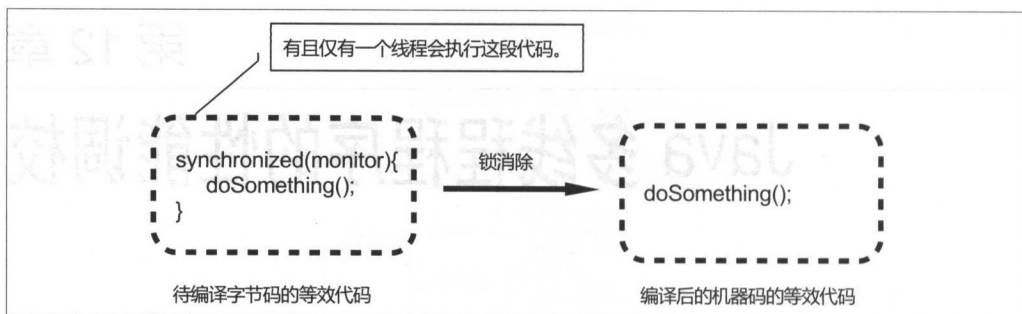


图 12-1 锁消除（Lock Elision）示意图

Java 标准库中的有些类（比如 `StringBuffer`）虽然是线程安全的，但是在实际使用中我们往往不在多个线程间共享这些类的实例。而这些类在实现线程安全的时候往往借助于内部锁。因此，这些类是锁消除优化的常见目标。在如清单 12-1 所示的例子中，JIT 编译器在编译 `toJSON` 方法的时候会将其调用的 `StringBuffer.append/toString` 方法内联（Inline）到该方法之中，这相当于把 `StringBuffer.append/toString` 方法的方法体中的指令复制到 `toJSON` 方法体之中。这里的 `StringBuffer` 实例 `sb` 是一个局部变量，并且该变量所引用的对象并没有被发布到其他线程，因此 `sb` 引用的对象只能够被 `sb` 所在的方法（`toJSON` 方法）的当前执行线程（一个线程）访问。所以，JIT 编译器此时可以消除 `toJSON` 方法中从 `StringBuffer.append/toString` 方法的方法体复制的指令所使用的内部锁。在这个例子中，`StringBuffer.append/toString` 方法本身所使用的锁并不会被消除，因为系统中可能还有其他地方在使用 `StringBuffer`，而这些代码可能会共享 `StringBuffer` 实例。

清单 12-1 可进行锁消除优化的示例代码

```

public class LockElisionExample {

    public static String toJSON(ProductInfo productInfo) {
        StringBuffer sbf = new StringBuffer();
        sbf.append("{\"productID\":\"").append(productInfo.productID);
        sbf.append("\", \"categoryID\":\"").append(productInfo.categoryID);
        sbf.append("\", \"rank\":\"").append(productInfo.rank);
        sbf.append("\", \"inventory\":\"").append(productInfo.inventory);
        sbf.append('}');

        return sbf.toString();
    }
}
  
```

锁消除优化所依赖的逃逸分析技术自 Java SE 6u23 起默认是开启的，但是锁消除优化

是在 Java 7 开始引入的³。

从上述例子可以看出，锁消除优化还可能要以 JIT 编译器的内联优化为前提。而一个方法是否会被 JIT 编译器内联取决于该方法的热度以及该方法对应的字节码的尺寸 (Bytecode Size)⁴。因此，锁消除优化能否被实施还取决于被调用的同步方法（或者带同步块的方法）是否能够被内联。

锁消除优化告诉我们在该使用锁的情况下必须使用锁，而不必过多在意锁的开销。开发人员应该在代码的逻辑层面考虑是否需要加锁，而至于代码运行层面上某个锁是否真的有必要使用则由 JIT 编译器来决定。锁消除优化并不表示开发人员在编写代码的时候可以随意使用内部锁（在不需要加锁的情况下加锁），因为锁消除是 JIT 编译器而不是 javac 所做的一种优化，而一段代码只有在其被执行的频率足够大的情况下才有可能会被 JIT 编译器优化⁵。也就是说在 JIT 编译器优化介入之前，只要源代码中使用了内部锁，那么这个锁的开销就会存在。另外，JIT 编译器所执行的内联优化、逃逸分析以及锁消除优化本身都是有其开销的。

在锁消除的作用下，利用 ThreadLocal 将一个线程安全的对象（比如 Random）作为一个线程特有对象来使用，不仅可以避免锁的争用，还可以彻底消除这些对象内部所使用的锁的开销。

12.1.2 锁粗化

锁粗化 (Lock Coarsening/Lock Merging) 是 JIT 编译器对内部锁的具体实现所做的一种优化，如图 12-2 所示。对于相邻的几个同步块，如果这些同步块使用的是同一个锁实例，那么 JIT 编译器会将这些同步块合并为一个大同步块，从而避免了一个线程反复申请、释放同一个锁所导致的开销。然而，锁粗化可能导致一个线程持续持有一个锁的时间变长，从而使得同步在该锁之上的其他线程在申请锁时的等待时间变长。例如在图 12-2 中，第 1

3 开启逃逸分析的虚拟机参数为“-XX:+DoEscapeAnalysis”，关闭逃逸分析的虚拟机参数为“-XX:-DoEscapeAnalysis”。注意：“-XX:”开头的虚拟机参数表示相应的参数是“不稳定的”，即 Oracle 公司可能会在不事先通知的情况下更改甚至废弃相应的参数。

4 比如，一个方法对应的字节码尺寸小于 35 字节（或者虚拟机参数“-XX:MaxInlineSize”所指定的参数值），或者该方法被调用的频率足够频繁（由 Java 虚拟机判定）并且方法对应的字节码尺寸小于 325 字节（或者虚拟机参数“-XX:MaxFreqInlineSize”所指定的参数值）。

5 比如一个方法或者循环体被执行了 10 000 次（或者虚拟机参数“-XX:CompileThreshold”所指定的参数值）以上。

一个同步块结束和第 2 个同步块开始之间的时间间隙中，其他线程本来是有机会获得 `monitorX` 的，但是经过锁粗化之后由于临界区的长度变长，这些线程在申请 `monitorX` 时所需的等待时间也相应变长了。因此，锁粗化不会被应用到循环体内的相邻同步块⁶。

相邻的两个同步块之间如果存在其他语句，也不一定就会阻碍 JIT 编译器执行锁粗化优化，这是因为 JIT 编译器可能在执行锁粗化优化前将这些语句挪到（即指令重排序）后一个同步块的临界区之中（当然，JIT 编译器并不会将临界区内的代码挪到临界区之外）。

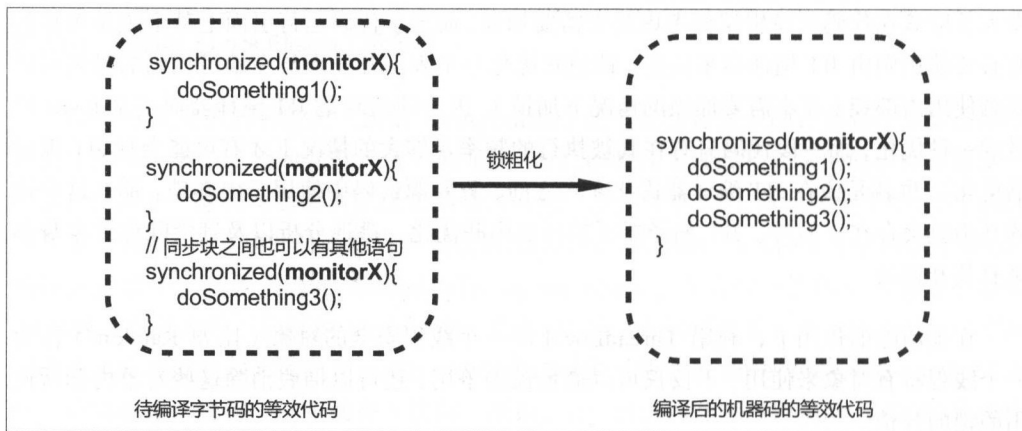


图 12-2 锁粗化（Lock Coarsening）示意图

实际上，我们写的代码中可能很少会出现图 12-2 中那种连续的同步块。这种同一个锁实例引导的相邻同步块往往是 JIT 编译器编译之后形成的。例如，在如清单 12-2 所示的例子中，`simulate` 方法连续调用 `randomIQ` 方法来生成 3 个符合正态分布（高斯分布）的随机智商（IQ）。在 `simulate` 方法被执行得足够频繁的情况下，JIT 编译器可能对该方法执行一系列优化：首先，JIT 编译器可能将 `randomIQ` 方法内联（inline）到 `simulate` 方法中，这相当于把 `randomIQ` 方法体中的指令复制到 `simulate` 方法之中。在此基础上，`randomIQ` 方法中的 `rnd.nextGaussian()` 调用也可能被内联，这相当于把 `Random.nextGaussian()` 方法体中的指令复制到 `simulate` 方法之中。`Random.nextGaussian()` 是一个同步方法，由于 `Random` 实例 `rnd` 可能被多个线程共享（因为 `simulate` 方法可能被多个线程执行），因此 JIT 编译器无法对 `Random.nextGaussian()` 方法本身执行锁消除优化，这使得被内联到 `simulate` 方法中的 `Random.nextGaussian()` 方法体相当于一个由 `rnd` 引导的同步块。经过上述优化之后，JIT 编译器便会发现 `simulate` 方法中存在 3 个相邻的由 `rnd`（`Random` 实例）引导的同步块，于

6 这其实也不是绝对的。因为 JIT 编译器也可能执行循环展开（Loop unroll）优化，该优化会使得循环语句消失。

是锁粗化优化便“粉墨登场”了。

清单 12-2 可进行锁粗化优化的示例代码

```
public class LockCoarseningExample {
    private final Random rnd = new Random();

    public void simulate() {
        int iq1 = randomIQ();
        int iq2 = randomIQ();
        int iq3 = randomIQ();
        act(iq1, iq2, iq3);
    }

    private void act(int... n) {
        // ...
    }

    // 返回随机的智商值
    public int randomIQ() {
        // 人类智商的标准差是 15, 平均值是 100
        return (int) Math.round(rnd.nextGaussian() * 15 + 100);
    }

    // ...
}
```

锁粗化默认是开启的。如果要关闭这个特性，我们可以在 Java 程序的启动命令行中添加虚拟机参数“-XX:-EliminateLocks”（开启则可以使用虚拟机参数“-XX:+EliminateLocks”）。

12.1.3 偏向锁

偏向锁（Biased Locking）是 Java 虚拟机对锁的实现所做的一种优化。这种优化基于这样的观测结果（Observation）：大多数锁并没有被争用（Contented），并且这些锁在其整个生命周期内至多只会被一个线程持有。然而，Java 虚拟机在实现 monitorenter 字节码（申请锁）和 monitorexit 字节码（释放锁）时需要借助一个原子操作（CAS 操作），这个操作代价相对来说比较昂贵。因此，Java 虚拟机会为每个对象维护一个偏好（Bias），即一个对象对应的内部锁第 1 次被一个线程获得，那么这个线程就会被记录为该对象的偏好线程（Biased Thread）。这个线程后续无论是再次申请该锁还是释放该锁，都无须借助原先（指未实施偏向锁优化前）昂贵的原子操作，从而减少了锁的申请与释放的开销。

然而，一个锁没有被争用并不代表仅仅只有一个线程访问该锁，当一个对象的偏好线程以外的其他线程申请该对象的内部锁时，Java 虚拟机需要收回（Revoke）该对象对原偏

好线程的“偏好”并重新设置该对象的偏好线程。这个偏好收回和重新分配过程的代价也是比较昂贵的，因此如果程序运行过程中存在比较多的锁争用的情况，那么这种偏好收回和重新分配的代价便会被放大。有鉴于此，偏向锁优化只适合于存在相当大一部分锁并没有被争用的系统之中。如果系统中存在大量被争用的锁而没有被争用的锁仅占极小的部分，那么我们可以考虑关闭偏向锁优化。

偏向锁优化默认是开启的。要关闭偏向锁优化，我们可以在 Java 程序的启动命令行中添加虚拟机参数“-XX:-UseBiasedLocking”（开启偏向锁优化可以使用虚拟机参数“-XX:+UseBiasedLocking”）。

12.1.4 适应性锁

适应性锁（Adaptive Locking，也被称为 Adaptive Spinning）是 JIT 编译器对内部锁实现所做的一种优化。

存在锁争用的情况下，一个线程申请一个锁的时候如果这个锁恰好被其他线程持有，那么这个线程就需要等待该锁被其持有线程释放。实现这种等待的一种保守方法我们在前面章节中已经介绍过——将这个线程暂停（线程的生命周期状态变为非 Runnable 状态）。由于暂停线程会导致上下文切换，因此对于一个具体锁实例来说，这种实现策略比较适合于系统中绝大多数线程对该锁的持有时间较长的场景，这样才能够抵消上下文切换的开销。另外一种实现方法就是采用忙等（Busy Wait）。所谓忙等相当于如下代码所示的一个循环体为空的循环语句：

```
// 当锁被其他线程持有时一直循环
while (lockIsHeldByOtherThread){}
```

可见，忙等是通过反复执行空操作（什么也不做）直到所需的条件成立为止而实现等待的。这种策略的好处是不会导致上下文切换，缺点是比较耗费处理器资源——如果所需的条件在相当长时间内未能成立，那么忙等的循环就会一直被执行。因此，对于一个具体的锁实例来说，忙等策略比较适合于绝大多数线程对该锁的持有时间较短的场景，这样能够避免过多的处理器时间开销。

事实上，Java 虚拟机也不是非要在上述两种实现策略之中择其一——它可以综合使用上述两种策略。对于一个具体的锁实例，Java 虚拟机会根据其运行过程中收集到的信息来判断这个锁是属于被线程持有时间“较长”的还是“较短”的。对于被线程持有时间“较长”的锁，Java 虚拟机会选用暂停等待策略；而对于被线程持有时间“较短”的锁，Java 虚拟机会选用忙等待策略。Java 虚拟机也可能先采用忙等待策略，在忙等失败的情况

下再采用暂停等待策略⁷。Java 虚拟机的这种优化就被称为适应性锁 (Adaptive Locking)，这种优化同样也需要 JIT 编译器介入。

适应性锁优化可以是以具体的一个锁实例为基础的。也就是说，Java 虚拟机可能对一个锁实例采用忙等待策略，而对另外一个锁实例采用暂停等待策略。

从适应性锁优化可以看出，内部锁的使用并不一定会导致上下文切换，这就是前面章节介绍锁与上下文切换时均说锁“可能”导致上下文切换的原因。

12.2 优化对锁的使用

接下来我们将从应用代码这个层次来探讨对锁的优化。尽管本节的例子我们会使用内部锁，但是这些优化对显式锁也是适用的。

12.2.1 锁的开销与锁争用监视

锁的开销包括以下几个方面。

- 上下文切换与线程调度开销。一个线程申请一个锁的时候，如果这个锁恰好被其他线程持有，那么该线程最终可能会被暂停。Java 虚拟机还需要为这些被暂停的线程维护一个等待队列（等待集），以便在这个锁被其持有线程释放的时候将这些线程唤醒。而线程的暂停与唤醒就是一个上下文切换的过程，并且 Java 虚拟机维护等待队列也会产生一定的开销。显然，非争用锁并不会导致上下文切换和等待队列的开销。
- 内存同步、编译器优化受限的开销。锁的内部实现所使用的内存屏障也会产生直接和间接的开销：直接的开销是内存屏障所导致的冲刷写缓冲器、清空无效化队列所导致的开销。另外，内存屏障会阻碍某些编译器优化。无论是争用锁还是非争用锁，都会产生这部分开销⁸。当然，非争用的锁如果最终适用锁消除优化的话，那么这个锁的任何开销都会被彻底消除。

7 这种情况下所使用的忙等待通常会设置一个循环被执行的次数限制或者时间限制。超过这个限制之后，锁仍然未申请成功，那么就认为忙等待失败。此时可以尝试暂停等待。

8 极端的非争用锁（锁的整个生命周期内只有一个线程访问的情况下）会被锁消除优化掉，因而这部分的开销也就不存在了。

- 限制可伸缩性。锁的排他性的本质是局部地将并发计算改为串行计算。这种特性会限制系统的可伸缩性。假设系统的某个操作每次执行的时候都需要申请一个锁，该锁平均被持有的时间为 5 毫秒，那么 1 秒之内该系统最多只能完成 200 个这样的操作，即这个系统的该操作的吞吐率为 200 TPS（Transaction per Second），无论这个系统有多少个处理器。可见，锁的排他性会导致处理器资源（以及其他资源）的浪费，并限制系统的吞吐率。

可见，锁的开销主要体现在争用锁（Contented Lock）上面。因此，减少锁的开销的一个基本思路就是消除锁的使用（使用锁的替代品）或者降低锁的争用程度。

影响锁的争用程度的因素有两个：程序申请锁的频率以及锁通常被持有的时间跨度。程序越是频繁地申请一个锁，或者这个锁通常被其持有线程持有的时间越长，那么这个锁的争用程度就越高；反之则该锁的争用程度就越低。

因此，降低锁的争用程度的基本思路就是尽可能地减少锁被持有的时间和（或）降低锁的申请频率。清单 12-3 所示的 Demo 展示了这一点。该 Demo 分别使用静态变量 `lockAccessFrequency` 和 `lockDuration` 来控制线程执行 `SharedResource.access()` 的频率以及 `SharedResource.access()` 的执行线程持有锁（`SharedResource` 当前实例）的时间跨度。

清单 12-3 锁争用 Demo

```
public class LockContentionDemo {
    // 用于模拟锁的持有时间跨度
    static long lockDuration = 100;
    static SharedResource sr = new SharedResource();
    // 用于模拟锁申请频率
    static long lockAccessFrequency = 50;

    public static void main(String[] args) throws InterruptedException {
        int argc = args.length;
        if (argc > 0) {
            lockDuration = Long.valueOf(args[0]);
        }
        if (argc > 1) {
            lockAccessFrequency = Long.valueOf(args[1]);
        }
    }

    int N = Runtime.getRuntime().availableProcessors();
    Thread[] threads = new Thread[N];
    for (int i = 0; i < threads.length; i++) {
        threads[i] = new Thread() {
            @Override
            public void run() {
                for (;;) {
                    sr.access();
                    try {

```

```

        Thread.sleep(lockAccessFrequency);
    } catch (InterruptedException ignored) {
    }
}
} // run 结束
};

} // for 结束

// 启动所有线程
Tools.startThread(threads);
Tools.delayedAction("The program will be terminated", new Runnable() {
    @Override
    public void run() {
        System.exit(0);
    }
}, 120);
}

static class SharedResource {
    public synchronized void access() {
        // 模拟实际操作耗时
        try {
            Thread.sleep(lockDuration);
        } catch (InterruptedException ignored) {
        }
    }
}
}
}
}

```

在不指定任何参数的情况下直接运行上述 Demo，并使用 JMC（Java Mission Control）可监视到此时该 Demo 对锁的争用情况，如图 12-3 所示。

可见，此时锁（SharedResource 当前实例）被争用的次数（Count）为 593，争用该锁的线程在申请锁时的平均等待时间（Average）为 646 毫秒。如果我们减少工作者线程对锁的持有时间跨度（lockDuration）或者降低工作者线程申请锁的频率（lockAccessFrequency），那么这些线程对锁的争用程度就会降低。例如，在笔者的实验环境下将 lockAccessFrequency 调整为 150，即降低工作者线程对锁的申请频率，此时锁被争用的次数为 598（略有上升），线程在申请锁时的平均等待时间（Average）为 149 毫秒（大为下降）；将 lockDuration 调整为 20，即减少工作者线程对锁的持有时间跨度，此时锁被争用的次数为 4，线程在申请锁时的平均等待时间为 1 毫秒⁹。可见，此时的锁争用几乎可以忽略。

9 对于锁争用监视，JMC 在数据采样的时候默认情况下只会记录那些等待锁超过 10 毫秒的等待事件。这里，笔者在启动 Flight Recording 时将其启动参数 Synchronization Threshold 设置为 1 毫秒。

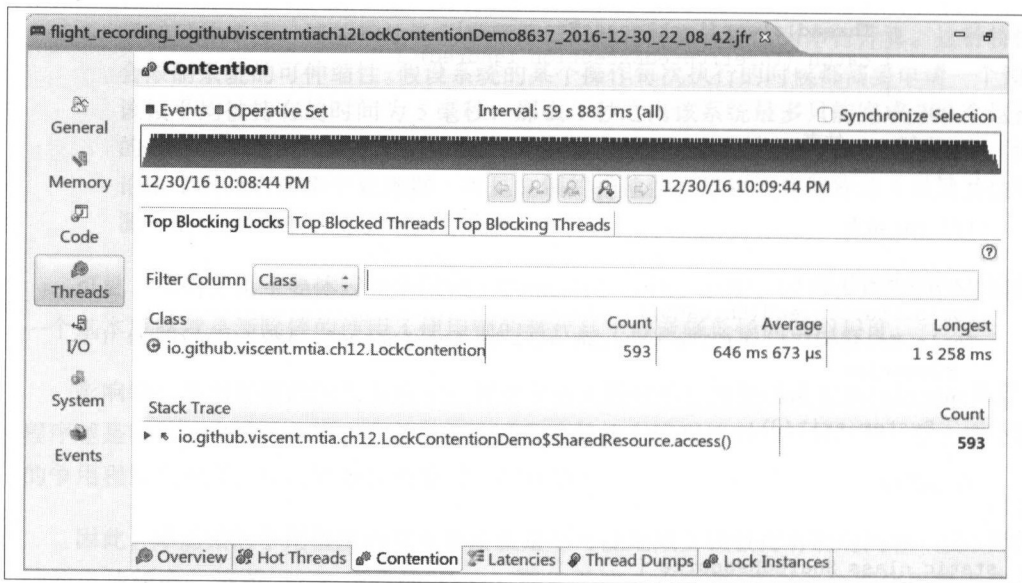


图 12-3 使用 JMC 监视锁争用情况

就具体实现而言，降低锁的争用程度可以从减小临界区长度以及减小锁的粒度这两个方面入手。另外，我们也可以从减少线程所需申请的锁的数量或者通过使用锁的替代品来减少乃至避免锁的开销。

12.2.2 使用可参数化锁

如果一个方法或者类内部锁使用的锁实例可以由该方法、类的客户端代码指定，那么我们就称这个锁是可参数化的，相应地，这个锁就被称为可参数化的锁。可参数化的锁在特定情况下有助于减少线程执行过程中参与的锁实例的个数，从而减少锁的开销。

Java 标准库中的抽象类 `java.io.Writer` 就使用了可参数化的锁。`Writer` 类内部维护了一个 `protected` 修饰的实例变量 `lock`，该变量充当了 `Writer.write/flush/close` 等方法所需的内部锁。`Writer` 类默认使用其子类的当前实例（`this` 关键字所代表的对象）作为 `lock` 的值，即 `Writer` 类默认使用的锁实例是其子类的当前实例。`Writer` 类的子类可以通过在其构造器中调用 `Writer` 的构造器 `Writer(Object lock)` 时指定一个对象或者直接在其构造器中为 `lock` 变量赋值的方式来设置 `Writer.write/flush/close` 等方法实际使用的锁实例。

下面我们看一个优化实战案例。某系统的某日志打印模块需要控制每个日志文件中最多可以包含 N （比如为 10000）条记录。该日志打印模块功能入口类如清单 12-4 所示，该

类的 `print` 方法用于向日志文件中写入一条日志记录，它封装了对单个日志文件中可包含的最大记录条数进行控制的逻辑。

清单 12-4 可改用可参数化锁的实例代码

```
public class LogPrinterV1 {
    final static SimpleDateFormat DATE_FORMAT = new SimpleDateFormat(
        "yyMMddHHmm");
    final static DecimalFormat DECIMAL_FORMAT = new DecimalFormat("00");
    final static int MAX_RECORDS_PER_FILE = 10_000;
    private PrintWriter pwr = null;
    private int recordsInFile = MAX_RECORDS_PER_FILE;
    private int fileSeq = 0;

    public void print(String record) {
        PrintWriter writer;
        try {
            synchronized (this) {
                writer = getPrintWriter();
                writer.println(record);
                recordsInFile++;
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public synchronized void shutdown() {
        if (null != pwr) {
            pwr.close();
            pwr = null;
        }
    }

    private PrintWriter getPrintWriter() throws IOException {
        PrintWriter writer = null;
        if (MAX_RECORDS_PER_FILE == recordsInFile) {
            String newFileName = retrieveFileName();
            writer = new PrintWriter(newFileName);
            recordsInFile = 0;
            if (null != pwr) {
                pwr.flush();
                pwr.close();
            }
            pwr = writer;
        } else {
```



```

        writer = pwr;
    }
    return writer;
}

protected String retrieveFileName() {
    String fileName;
    fileName = "/home/viscent/tmp/logs/" + DATE_FORMAT.format(new Date())
        + DECIMAL_FORMAT.format(fileSeq) + ".log";
    if (++fileSeq > 99) {
        fileSeq = 0;
    }
    return fileName;
}
}

```

显然，一个日志文件在还没有被写满（即记录中的文件条数达到最大记录条数）的时候，任何一个线程都可以往其中写入记录（通过执行上述 `print` 方法）。因此，上述 `print` 方法使用的 `PrintWriter` 实例实际上是共享的。尽管如此，但由于这个 `PrintWriter` 实例只能通过 `print` 和 `shutdown` 方法来访问，因此即便 `PrintWriter` 类不是线程安全的（实际上它是线程安全的），我们只需要让 `print` 和 `shutdown` 方法使用同一个锁（正如清单 12-4 那样）实例也就可以保障线程安全了。然而，`PrintWriter` 内部又确实实地使用了另外一个锁实例——`PrintWriter` 类的当前实例。为了减少锁的开销，此时我们可以将上述 `PrintWriter` 实例所使用的锁实例指定为 `print` 和 `shutdown` 方法自身所使用的锁实例——`LogPrinterV1` 类的当前实例。这样一来，`print`、`shutdown` 以及 `PrintWriter` 实例的各个方法（`println`、`flush` 和 `close`）仅使用一个锁实例，如清单 12-5 所示。

清单 12-5 使用可参数化锁的实例代码

```

public class LogPrinterV2 {
    final static SimpleDateFormat DATE_FORMAT = new SimpleDateFormat(
        "yyMMddHHmm");
    final static DecimalFormat DECIMAL_FORMAT = new DecimalFormat("00");
    final static int MAX_RECORDS_PER_FILE = 10_000;
    private PrintWriter pwr = null;
    private int recordsInFile = MAX_RECORDS_PER_FILE;
    private int fileSeq = 0;

    public void print(String record) {
        PrintWriter writer;
        try {
            synchronized (this) {
                writer = getPrintWriter();
                writer.println(record);
            }
        }
    }
}

```

```

        recordsInFile++;
    }

    } catch (Exception e) {
        e.printStackTrace();
    }
}

public synchronized void shutdown() {
    if (null != pwr) {
        pwr.close();
        pwr = null;
    }
}

private PrintWriter getPrintWriter() throws IOException {
    PrintWriter writer = null;
    if (MAX_RECORDS_PER_FILE == recordsInFile) {
        String newFileName = retrieveFileName();
        writer = new PrintWriter(newFileName) {
            {
                lock = this;
            }
        };
        recordsInFile = 0;
        if (null != pwr) {
            pwr.flush();
            pwr.close();
        }
        pwr = writer;
    } else {
        writer = pwr;
    }
    return writer;
}

protected String retrieveFileName() {
    // .....
}
}

```

在优化后的代码中，在 `getPrintWriter` 方法中我们不直接创建 `PrintWriter` 实例，而是创建一个 `PrintWriter` 类的匿名子类并在该子类的实例初始化块（`{}`）中将该实例所使用的锁实例 `lock` 设置为 `this`（`LogPrinterV2` 类的当前实例）。经过这种改造，`print` 方法执行过程中只需要申请一个锁——`this` 关键字所代表的 `LogPrinterV2` 类当前实例，而不再像清单 12-4 那样需要申请两个锁（`print` 方法所属的实例自身以及 `PrintWriter` 实例）。此时，尽管

print 方法执行过程中调用的 `PrintWriter.flush/close/println` 等方法仍然是带同步块的方法，但是由于锁的可重入性，print 方法的执行线程在已经持有 this 所代表的锁的情况下重新申请/释放这个锁的开销已经降低了不少。

从面向对象编程的角度来看，使用可参数化锁一定程度上破坏了封装性。假如指定的锁实例被其他代码不恰当地使用了，那么可参数化锁的使用可能会增加锁的争用。

12.2.3 减小临界区的长度

减小临界区的长度可以减少锁被持有的时间从而降低锁被争用的概率，这有利于减少锁的开销。另外，减少锁的持有时间有利于 Java 虚拟机的适用性锁优化发挥作用：在多数线程持有锁的时间都很短的情况下，锁的申请线程可以通过忙等而无须通过暂停线程来等待被争用的锁的释放，这有利于减少上下文切换开销。

临界区逻辑上连贯的一些操作往往可以划分为几个部分：预处理操作（Pre-process）、共享变量访问操作以及后处理操作（Post-process）。其中，预处理操作和后处理操作往往是不涉及共享变量的访问的，因此把这两种操作挪到临界区之外可以在不导致线程安全问题的前提下减小临界区的长度。如果预处理操作、后处理操作中涉及 I/O 操作、阻塞操作等比较耗时的操作，那么将这些操作挪到临界区之外可以有效地减少锁被持有的时间。

下面我们看一个优化实战案例。某电信系统需要往某个目录（目标目录）中动态写入一批文件¹⁰。为了限制这些文件的个数，我们规定这些文件需要被写入目标目录的子目录（目标子目录）中，子目录的个数最多为 M （比如为 100）个，并且每个子目录中最多只能够包含 N （比如 2000）个文件。当需要的子目录个数超过 M 时，最老的子目录以及其中的所有文件会被删除。清单 12-6 中的 `apply4Filename` 方法用于获取待写文件的文件名以及这个文件的目标子目录名，该同步方法虽然满足了上述要求，但是其临界区的长度是可以减小的。

清单 12-6 临界区长度可减小的实例代码

```
public class SectionBasedStorageV1 {
    private Deque<String> sectionNames = new LinkedList<String>();
    // Key->value: 存储子目录名->子目录下缓存文件计数器
    private Map<String, AtomicInteger> sectionFileCountMap = new HashMap<>();
    private int maxFilesPerSection = 2000;
    private int maxSectionCount = 100;
    private String storageBaseDir = System.getProperty("java.io.tmpdir") + "/vpn";
```

10 所谓“动态”指的是待写文件的个数事先未知。

```

public SectionBasedStorageV1() {
    File dir = new File(storageBaseDir);
    if (!dir.exists()) {
        dir.mkdirs();
    }
}

public synchronized String[] apply4Filename() {
    String sectionName;
    int iFileCount;
    String[] fileName = new String[2];
    // 获取当前的存储子目录名
    sectionName = getSectionName();
    AtomicInteger fileCount;
    fileCount = sectionFileCountMap.get(sectionName);
    iFileCount = fileCount.get();
    // 当前存储子目录已满
    if (iFileCount >= maxFilesPerSection) {
        if (sectionNames.size() >= maxSectionCount) {
            // 删除最老的存储子目录
            String oldestSectionName = sectionNames.removeFirst();
            removeSection(oldestSectionName);
        }
        // 创建新的存储子目录
        sectionName = makeNewSectionDir();
        fileCount = sectionFileCountMap.get(sectionName);
    }
    iFileCount = fileCount.incrementAndGet();
    fileName[0] = storageBaseDir + "/" + sectionName + "/"
        + new DecimalFormat("0000").format(iFileCount) + "-"
        + new Date().getTime() / 1000 + ".rq";
    fileName[1] = sectionName;
    return fileName;
}

public void decrementSectionFileCount(String sectionName) {
    AtomicInteger fileCount = sectionFileCountMap.get(sectionName);
    if (null != fileCount) {
        fileCount.decrementAndGet();
    }
}

private boolean removeSection(String sectionName) {
    boolean result = true;
    File dir = new File(storageBaseDir + "/" + sectionName);
    for (File file : dir.listFiles()) {
        result = result && file.delete();
    }
}

```

```

    }
    result = result && dir.delete();
    return result;
}

private String getSectionName() {
    String sectionName;
    if (sectionNames.isEmpty()) {
        sectionName = makeNewSectionDir();
    } else {
        sectionName = sectionNames.getLast();
    }
    return sectionName;
}

private String makeNewSectionDir() {
    String sectionName;
    SimpleDateFormat sdf = new SimpleDateFormat("MMddHHmmss");
    sectionName = sdf.format(new Date());
    File dir = new File(storageBaseDir + "/" + sectionName);
    if (dir.mkdir()) {
        sectionNames.addLast(sectionName);
        sectionFileCountMap.put(sectionName, new AtomicInteger(0));
    } else {
        throw new RuntimeException("Cannot create section dir " + sectionName);
    }
    return sectionName;
}
}

```

`apply4Filename` 方法所执行的操作可以分解为这样一个操作：其预处理操作为空；其共享变量访问操作为获取待写文件的目标子目录名（`sectionName`）以及目标子目录中的现有文件个数；其后处理操作为构造文件名并在目标目录满（子目录个数达到 M ）时删除最老的子目录。

因此，我们可以把 `apply4Filename` 方法的后处理操作挪到临界区之外，以减小临界区的长度但不会影响线程安全，如清单 12-7 所示。

清单 12-7 减小 `apply4Filename` 方法的临界区长度

```

public class SectionBasedStorageV2 {
    private Deque<String> sectionNames = new LinkedList<String>();
    // Key->value: 存储子目录名->子目录下缓存文件计数器
    private Map<String, AtomicInteger> sectionFileCountMap = new HashMap<>();
    private int maxFilesPerSection = 2000;
    private int maxSectionCount = 100;
}

```

```

private String storageBaseDir = System.getProperty("java.io.tmpdir") + "/vpn";

public String[] apply4Filename() {
    String sectionName;
    int iFileCount;
    String[] fileName = new String[2];
    String oldestSectionName = null;
    synchronized (this) {
        // 获取当前的存储子目录名
        sectionName = getSectionName();
        AtomicInteger fileCount;
        fileCount = sectionFileCountMap.get(sectionName);
        iFileCount = fileCount.get();
        // 当前存储子目录已满
        if (iFileCount >= maxFilesPerSection) {
            if (sectionNames.size() >= maxSectionCount) {
                oldestSectionName = sectionNames.removeFirst();
            }
            // 创建新的存储子目录
            sectionName = makeNewSectionDir();
            fileCount = sectionFileCountMap.get(sectionName);
        }
        iFileCount = fileCount.incrementAndGet();
    } // 临界区结束
    fileName[0] = storageBaseDir + "/" + sectionName + "/"
        + new DecimalFormat("0000").format(iFileCount) + "-"
        + new Date().getTime() / 1000 + ".rq";
    fileName[1] = sectionName;
    if (null != oldestSectionName) {
        // 删除最老的存储子目录
        removeSection(oldestSectionName);
    }
    return fileName;
}

public void decrementSectionFileCount(String sectionName) {
    AtomicInteger fileCount = sectionFileCountMap.get(sectionName);
    if (null != fileCount) {
        fileCount.decrementAndGet();
    }
}
// 省略与清单 12-6 相同的方法
}

```

相比于清单 12-6 中的 `apply4Filename` 方法，新的 `apply4Filename` 方法在临界区中仅判断并记录（通过局部变量 `oldestSectionName`）是否需要删除最老的子目录，而具体删除最老的子目录这个 I/O 操作则是在临界区外执行的。另外，构造目标文件的文件名也是放

在临界区之外进行的。当然，将最老的子目录删除这个操作移动到临界区之外，会导致在某一个瞬间目标目录中的子目录个数可能超过 M （允许的子目录的最大个数），不过这一点在该系统中是可接受的。

12.2.4 减小锁的粒度

降低锁的争用程度的另外一种思路是降低锁的申请频率。而减小锁的粒度可以降低锁的申请频率，从而减小锁被争用的概率。减小锁粒度的一种常见方法是将一个粒度较粗的锁拆分成若干粒度更细的锁，其中每个锁仅负责保护（Guard）原粗粒度锁所保护的所有共享变量中的一部分共享变量，如图 12-4 所示。这种技术被称为锁拆分技术（Lock Splitting）。

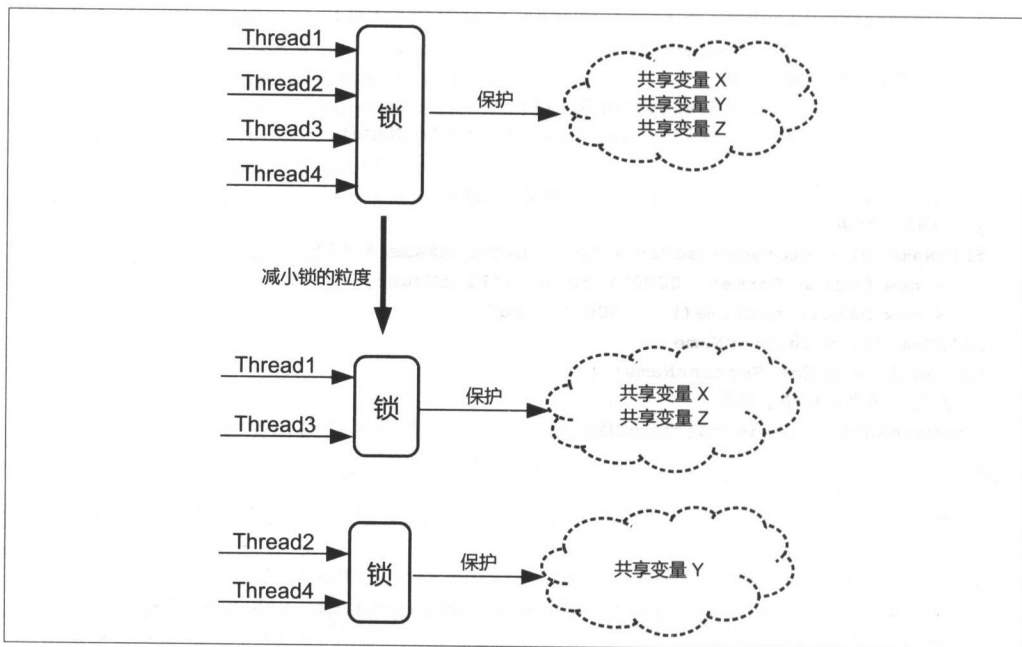


图 12-4 通过拆分锁减小锁的粒度示意图

下面看一个锁拆分优化实战实例。某分布式系统在调用其下游部件的服务时会通过 Socket 网络连接给对方发送一个请求消息（RequestMessage），并将该请求消息注册到请求注册表（RequestRegistry）中。每个请求消息都有唯一的编号（Request ID）。发送请求消息的线程（请求消息发送线程）会通过等待（Object.wait(long)调用）来获取由下游部件发送过来的与这个请求相对应的响应消息（ResponseMessage）。该系统会专门启动若干工

作者线程（响应消息接收线程）用于接收来自下游部件的响应消息。每个响应消息唯一对应一个请求消息。响应消息接收线程在接收到响应消息后，会根据消息中包含的 Request ID 到请求注册表查找相应的请求消息。若找到相应的请求消息，则将该响应消息关联到对应的请求消息上，并通知请求消息发送线程。这样的功能是在 Java 1.4 时代实现的¹¹。

清单 12-8 展示了请求注册表的一个不成熟实现代码。

清单 12-8 可进行锁拆分优化的类 NaiveRequestRegistry

```
public class NaiveRequestRegistry implements RequestRegistry {
    @SuppressWarnings({ "rawtypes" })
    private final Map/* <String, RequestMessage> */requests = new HashMap();

    // 注册请求消息
    @SuppressWarnings("unchecked")
    @Override
    public synchronized void registerRequest(RequestMessage request) {
        String requestID = request.getID();
        requests.put(requestID, request);
    }

    // 取消请求消息注册
    @Override
    public synchronized void unregisterRequest(RequestMessage request) {
        String requestID = request.getID();
        requests.remove(requestID);
    }

    // 请求发送线程可调用该方法等待指定请求消息对应的响应消息
    @Override
    public synchronized ResponseMessage
        waitForResponse(RequestMessage request, long timeout)
        throws TimeoutException, InterruptedException {
        ResponseMessage res = null;
        long start = System.currentTimeMillis();
        long waitTime;
        long now;
        boolean isTimedout = false;
        while (null == (res = request.getResponse())) {
            now = System.currentTimeMillis();
            // 计算剩余等待时间
            waitTime = timeout - (now - start);
            if (waitTime <= 0) {
                // 等待超时退出
                isTimedout = true;
            }
        }
    }
}
```

¹¹ 那时 Java 标准库中还没有 ConcurrentHashMap、Condition 这样的类/接口。


```

        break;
    }
    wait(waitTime);
} // while 循环结束
if (isTimedout) {
    unregisterRequest(request);
    throw new TimeoutException(timeOut, request.toString());
}
return res;
}

// 响应消息接收线程接收到消息后会调用该方法
@Override
public synchronized void responseReceived(ResponseMessage response) {
    String requestID = response.getRequestID();
    RequestMessage request = (RequestMessage) requests.get(requestID);
    // request 为 null, 说明响应没有在规定时间内到达当前系统
    if (null != request) {
        requests.remove(requestID);
        request.setResponse(response);
        notifyAll();
    }
}
}

```

`RequestRegistry.registerRequest` 方法用于注册指定的请求消息。请求消息发送线程可通过调用 `RequestRegistry.waitForResponse` 方法来获取指定请求消息对应的响应消息。响应消息接收线程在接收到下游部件发送过来的响应消息后，会调用 `RequestRegistry.responseReceived` 方法将相应的响应消息关联到对应的请求消息上，并通知请求发送线程。

`NaiveRequestRegistry` 类中的所有方法都是同步方法。这就意味着当响应消息接收线程将其接收到的一个响应消息关联到对应的请求消息之上的时候（`RequestRegistry.responseReceived` 方法调用），请求发送线程无法注册新的请求消息（`RequestRegistry.registerRequest` 调用），因而也就无法发送新的请求消息。显然，这限制了请求发送线程以及响应消息接收线程各自的吞吐率（并发性降低）。我们可以利用锁拆分技术对 `NaiveRequestRegistry` 类中使用锁（`NaiveRequestRegistry` 类的当前实例）进行优化，如清单 12-9 所示。

清单 12-9 对 `NaiveRequestRegistry` 类进行拆分锁优化

```

public class FineRequestRegistry implements RequestRegistry {
    @SuppressWarnings({ "rawtypes" })
    private final Map/* <String, RequestMessage> */requests = new HashMap<>();
}

```

```

@SuppressWarnings("unchecked")
@Override
public synchronized void registerRequest(RequestMessage request) {
    String requestID = request.getID();
    requests.put(requestID, request);
}

@Override
public synchronized void unregisterRequest(RequestMessage request) {
    String requestID = request.getID();
    requests.remove(requestID);
}

@Override
public ResponseMessage waitForResponse(RequestMessage request, long timeOut)
    throws TimeoutException, InterruptedException {
    ResponseMessage res = null;
    long start = System.currentTimeMillis();
    long waitTime;
    long now;
    boolean isTimedout = false;
    synchronized (request) {
        while (null == (res = request.getResponse())) {
            now = System.currentTimeMillis();
            // 计算剩余等待时间
            waitTime = timeOut - (now - start);
            if (waitTime <= 0) {
                // 等待超时退出
                isTimedout = true;
                break;
            }
            request.wait(waitTime);
        } // while 循环结束
    } // synchronized 结束
    if (isTimedout) {
        unregisterRequest(request);
        throw new TimeoutException(timeOut, request.toString());
    }
    return res;
}

@Override
public void responseReceived(ResponseMessage response) {
    String requestID = response.getRequestID();
    RequestMessage request = null;
    synchronized (this) {
        request = (RequestMessage) requests.get(requestID);
        if (null == request) {

```

```

        return;
    }
    requests.remove(requestID);
}
synchronized (request) {
    request.setResponse(response);
    request.notify();
}
}
}

```

在优化后的代码中，`waitForResponse` 方法在等待响应消息的时候不再调用 `RequestRegistry.wait(long)` 而是调用 `Request.wait(long)`，因此该方法内部使用一个由 `Request` 实例引导的同步块即可，而无须将方法本身定义为同步方法。这个更改使得 `responseReceived` 方法也无须是同步方法：`responseReceived` 方法内部有一个由 `Request` 实例引导的同步块，我们在该同步块中调用 `Request.notify()` 即可实现通知请求消息发送线程。而 `registerRequest/unregisterRequest` 方法仍然是同步方法。此时 `registerRequest/unregisterRequest` 方法使用的锁是 `FineRequestRegistry` 的当前实例，该锁所保护的共享数据是实例变量 `requests` (`HashMap`)，而 `waitForResponse/responseReceived` 方法使用的锁主要是 `Request` 实例，该锁所保护的共享数据是具体的请求消息。而原来的代码（清单 12-8）中使用的锁（`NaiveRequestRegistry` 的当前实例）所保护的共享数据既包含实例变量 `requests` 也包含具体的请求消息，因此，相比原先使用的一个粒度较粗的锁（它保护两种共享数据），现在使用的两个粒度更细的锁（这两个锁各自仅保护一种共享数据）相当于减小了原先锁的粒度，从而降低了锁被争用的概率，即降低了 `registerRequest/unregisterRequest` 方法的执行线程（请求消息发送线程）和 `waitForResponse/responseReceived` 方法的执行线程（响应消息接收线程）争用同一个锁的概率，这有利于提高这两种线程的并发性，从而提高各自的吞吐率。

当然，就性能而言，如果我们将该案例中的实例变量 `requests` 的类型改为 `ConcurrentHashMap`，那么请求消息发送线程和响应消息接收线程的并发性将会进一步提升。这里我们并不这么做，一方面是因为当时（案例真实代码产生之时）Java 标准库并无 `ConcurrentHashMap` 类，另一方面主要是为了突出的锁拆分优化本身。

对于高争用的锁来说，锁拆分带来的效果可能并不是那么明显。这就好比银行在营业厅中出现一个非常忙碌的柜台时新开一个柜台来“分流”排队客户所带来的效果——一个非常忙碌的柜台变成两个忙碌的柜台。

锁拆分这种技术可以演进为另外一种被称为锁分段的技术。锁分段（Lock Striping）

是指对同一个数据结构内不同部分的数据使用不同锁实例进行加锁的技术。ConcurrentHashMap 内部就使用了锁分段技术, 如图 12-5 所示。ConcurrentHashMap 内部会创建 N (默认值为 16) 个锁实例。以 put 操作为例, 一个线程执行 put 方法时提供的 key 参数对应的 HashCode (即 `key.hashCode()` 返回值) 会传递给一个 Hash 函数, 该函数的返回值介于 0 与 $N-1$ 之间。ConcurrentHashMap 通过该 Hash 函数的返回值就能够确定当前线程需要使用的锁实例。因此, 同时执行 put 操作的不同线程只要其提供的 key 值不一样, 那么它们所需要使用的锁实例也可能是不一样的。这就使得一个锁实例可以保护多个桶 (Bucket) 中的条目。因此, 相对于 HashTable 和 Collections.synchronizedMap 方法返回的同步对象这些内部使用一个锁实例来保护整个数据结构而言, ConcurrentHashMap 内部所使用的这些锁的粒度已经小了许多。在这种加锁方式下, 假设有 N 个线程同时执行 put 操作, 并且这些线程所提供的 key 值能够使上述 Hash 函数的返回值各不相同 (也就是说这些线程所提供的 key 值至少应是各不相同的), 那么这些线程分别使用的是不同的锁实例, 即它们之间不存在锁争用。这就是我们在第 6 章讲 ConcurrentHashMap 默认情况下可以支持 16 ($N=16$) 个并发更新线程的原因。

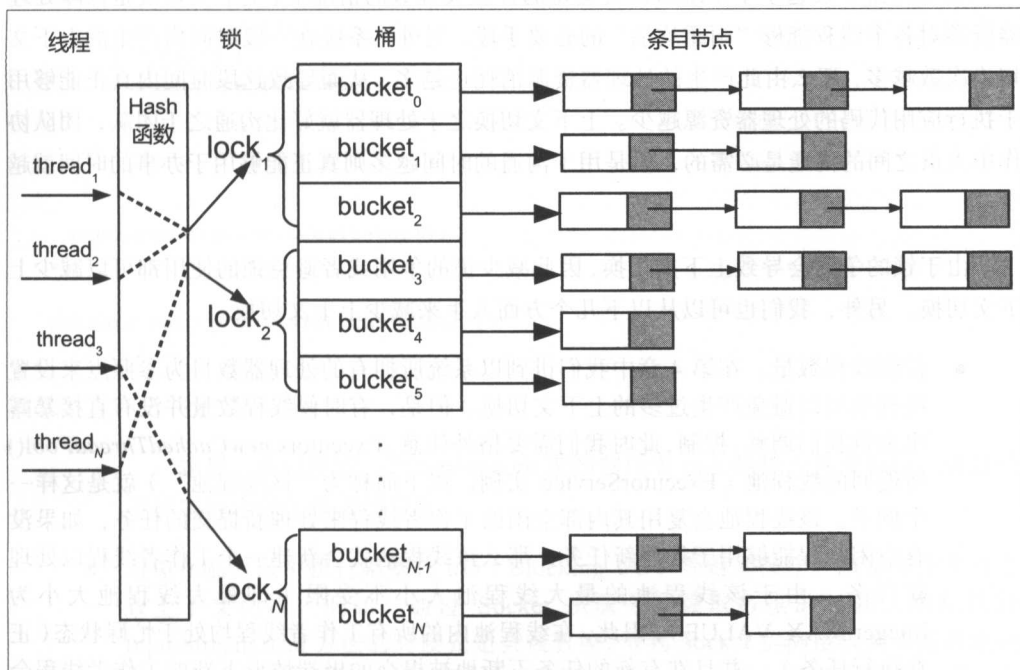


图 12-5 锁分段在 ConcurrentHashMap 中的应用示意图

锁分段会使对整个对象进行加锁变得困难甚至于不可能。例如, 要对整个 HashTable

实例进行加锁，我们只需要使用“synchronized(HashMap 实例)”这样的语句，而同样的方法无法实现在 ConcurrentHashMap 外部对整个 ConcurrentHashMap 实例进行加锁。即使是在 ConcurrentHashMap 内部，如果要对整个实例进行加锁的效果，那么由于 Java 语言本身并不支持申请多个数量可变的锁（即一个线程申请 N 个锁，而 N 的值在编译阶段不可知），因此我们需要通过递归算法才能够实现¹²。

12.2.5 考虑锁的替代品

在条件允许的情况下，我们也可以考虑使用锁的替代品来避免锁的开销和问题。这些有条件替代品包括：volatile 关键字（参见第 3 章）、原子变量（参见第 3 章）、无状态对象（参见第 6 章）、不可变对象（参见第 6 章）和线程特有对象（参见第 6 章）。

12.3 减少系统内耗：上下文切换

在处理器个数远小于系统所需要支持的并发线程数的情况下，上下文切换是保障处理器资源对各个线程能够“雨露均沾”的必要手段。另外，系统在一段时间内产生的上下文切换次数越多，那么由此产生的处理器资源消耗也越多，从而导致这段时间内真正能够用于执行应用代码的处理器资源越少。上下文切换之于处理器就好比沟通之于团队，团队协作中人员之间的沟通是必需的，但是用于沟通的时间越多则真正能够用于办事的时间就越少。

由于锁的争用会导致上下文切换，因此减少锁的争用或者避免锁的使用都可以减少上下文切换。另外，我们也可以从以下几个方面入手来减少上下文切换。

- 控制线程数量。在第 4 章中我们讲到以系统所拥有的处理器数目为参照点来设置线程数可以避免产生过多的上下文切换。但是，有时候线程数量并没有直接暴露出来供我们调整、控制，此时我们需要格外注意。Executors.newCachedThreadPool() 所返回的线程池（ExecutorService 实例，以下简称为“该线程池”）就是这样一个例子。该线程池会复用其内部空闲的工作者线程来处理新提交的任务，如果没有空闲线程能够用于处理新任务，那么该线程池就会新建一个工作者线程以处理新任务。由于该线程池的最大线程池大小不受限（即最大线程池大小为 Integer.MAX_VALUE），因此，在线程池内的所有工作者线程均处于忙碌状态（正在执行任务），并且在有新的任务不断地被提交的极端情形下新的工作者线程会不断被创建，最终导致该线程池的工作者线程总数远远超过系统所拥有的处理器

12 参见：<https://www.ibm.com/developerworks/library/j-jtp08223/>。

数目，从而导致过多的上下文切换而拖慢了整个系统（包括这个系统所运行的其他非 Java 应用程序）。因此，该线程池适合用来处理大量耗时较短的仅涉及非阻塞操作的任务。而如果要使用该线程池来处理耗时较长甚至于涉及阻塞操作的任务，那么就需要格外注意控制该线程池的实际工作者线程总数。由于该线程池并没有提供接口来控制其最大线程池的大小，因此我们无法直接控制工作者线程数上限，而需要借助 Semaphore 来达到控制并发线程数的目的，如清单 12-10 所示。

清单 12-10 控制 Executors.newCachedThreadPool()返回的线程池的工作者线程数示例代码

```
public class ImplicitControlThreadsCount {
    final ExecutorService executorService = Executors.newCachedThreadPool();
    final Semaphore semaphore = new Semaphore(Runtime.getRuntime()
        .availableProcessors() * 2);

    public void doSomething(final String data) throws InterruptedException {
        semaphore.acquire();
        Runnable task = new Runnable() {
            @Override
            public void run() {
                try {
                    process(data);
                } finally {
                    semaphore.release();
                }
            }
        };
        executorService.submit(task);
    }

    private void process(String data) {
        // .....
    }
}
```

这里，我们使用一个 Semaphore 来控制任务 task 的提交并发程度，从而限制了线程池中的工作者线程数。

- 避免在临界区中执行阻塞式 I/O（Blocking I/O）等阻塞操作。阻塞操作本身会导致上下文切换。例如，通过 Socket 从服务器读取数据的时候，线程执行到 InputStream.read 方法的时候先是会被暂停，等到 Socket 实例接收到服务器返回的数据的时候这个线程才被唤醒¹³。显然，这是一个上下文切换的过程。当一个线程因执行临界区中的阻塞操作而被暂停的时候，这个线程所持有的引导该临界区

13 其中的 InputStream 实例是通过 Socket.getInputStream()调用获取的。

的锁并没有被释放。这就可能导致其他线程申请这个锁的时候，该锁仍然还被这个被暂停的线程所持有。因此，临界区中的阻塞操作会增加引导这个临界区的锁被争用的可能性。而被争用的锁又可能导致上下文切换，因此在临界区中执行阻塞操作会进一步增加上下文切换。避免在临界区中执行阻塞式 I/O 等阻塞操作的典型技巧是在多线程环境中特意使用单线程来执行 I/O 操作。例如，某系统的一个数据同步模块需要将本地的一批文件（文件数量事先未知）上传到指定的 FTP 服务器。尽管这个模块实现 FTP 上传功能时使用的是一个非线程安全的组件，但是我们并没有对这个组件的使用进行加锁，而是把需要上传的文件（`java.io.File` 实例）存入一个队列，并专门设置一个工作者线程（仅一个线程）从该队列中取出文件并将其上传到 FTP 服务器。在这种设计中，尽管队列本身会涉及锁的使用，但是由于执行 FTP 上传文件这个网络 I/O 阻塞操作的线程只有一个，因此这不仅实现了线程安全（单个线程访问非线程安全的 FTP 组件并不会产生线程安全问题），还避免了由于阻塞操作（FTP 上传）可能增加的上下文切换。

- 避免在临界区中执行比较耗时的操作。在临界区中执行比较耗时的操作会增加引导该临界区的锁的持有时间，从而增加这个锁被争用的概率。而被争用的锁可能导致上下文切换。因此，在临界区中执行比较耗时的操作也会增加上下文切换的可能性。
- 减少 Java 虚拟机的垃圾回收。Java 垃圾回收器的运行可能导致 Stop-the-World 事件，即所有应用线程被暂停的现象。Java 垃圾回收器（Garbage Collector）在其工作过程中往往需要移动存活对象（Live Object，即未被垃圾回收掉的对象）。例如，在次要垃圾回收（Minor Garbage Collection）过程中垃圾回收器需要将存活对象从年轻代（Young Generation）移动到年老代（Old Generation）；为避免内存碎片问题而进行的内存整理（Compact）也涉及存活对象的移动。由于移动存活对象意味着这些对象所在的内存地址发生变化，因此在移动存活对象前垃圾回收器需要将所有应用线程暂停，并在移动结束后再将所有应用线程唤醒。因此，减少 Java 虚拟机垃圾回收的频率可以减少上下文切换。

12.4 多线程编程的“三十六计”：多线程设计模式

设计模式是从实践中总结出来的软件设计中给定背景（Context）下普遍存在的问题的一般性可复用解决方案。作为一种可复用解决方案，设计模式有点类似于组织（比如公司、项目组）为了达成其目标而采取的解决方案：设置一定的角色（比如管理人员、工程师），组织内担当各个角色的人员各司其职、相互协作，从而完成整个组织的目标。特定的设计模式都是通过其特定的参与者（Participant，相当于上述例子中的“角色”）以及这些参与者间的互相协作来解决特定问题的。

多线程设计模式则是设计模式在多线程编程中的一种应用。本书前面章节其实已经介绍过一些多线程设计模式。例如，第 5 章我们介绍过生产者—消费者模式，第 6 章介绍的 ThreadLocal 相当于 Thread Specific Storage（线程特有存储）模式的一个具体实现，第 9 章介绍的 Future 接口和 FutureTask 相当于 Promise（承诺）模式的一个具体实现，第 9 章介绍的 ThreadPoolExecutor 相当于 Thread Pool（线程池）模式的一个具体实现。

生产者—消费者模式的参与者包括 Product（产品）、Producer（生产者）、Consumer（消费者）和 Channel（传输通道），这些参与者的职责与协作情况是：Producer 仅负责生产 Product 并将其存入 Channel，而 Consumer 仅负责从 Channel 中取出 Product 进行加工（消费），Producer 与 Consumer 间不直接通信而是通过 Channel 传输数据（产品）。因此，Producer 实例和 Consumer 实例可以各自运行在不同的线程之中从而提高并发性。例如，在第 4 章的第 2 个实战案例（响应延时统计程序，代码见清单 4-7）中，我们通过使用生产者—消费者模式不仅得益于并发性的提高而将程序的处理能力相对于单线程程序提升 1 倍多，而且相对于未使用设计模式的“一般”多线程程序还简化了程序的算法（不用考虑表示一对请求和响应的记录被写入两个日志文件之中这种边界情形）。

多线程设计模式为多线程编程实现其目标——提高并发性提供了指引，恰当地应用多线程设计模式不仅可以提高程序的性能、可伸缩性，某些情况下还可能简化程序。当然，即使是应用了多线程设计模式的多线程程序，其相比于功能上等效的单线程程序而言还是更为复杂。

有关多线程设计模式的进一步内容可以参考本系列图书的“设计模式篇”¹⁴。

12.5 性能的隐形杀手：伪共享

由于一个缓存行中可以存储多个变量的副本，因此即便是在两个线程各自仅访问各自的共享变量（它们之间不存在共同的共享变量）的情况下，一个线程更新其共享变量也可能导致另外一个线程访问其共享变量时产生缓存未命中，这种现象就被称为伪共享（False Sharing）。

在多个线程访问同一组共享变量的情况下，一个处理器上的线程更新了其中一个共享变量，会导致其他处理器上包含这个共享变量副本的缓存条目被无效化（Invalidated），即相应缓存条目的状态被置为 I；因此，这些处理器上运行的其他线程再次访问（包括读和

¹⁴ 即《Java 多线程编程实战指南（设计模式篇）》（标准书号：ISBN 978-7-121-27006-2，电子工业出版社出版）。

写) 这个被无效化的缓存条目的缓存行中曾经存有副本的任何一个共享变量时, 都会产生缓存未命中 (Cache Miss)。如图 12-6 所示, 假设两个线程 thread_1 和 thread_2 分别运行在处理器 Processor 0 和 Processor 1 上, x 和 y 是内存地址上相邻的 (就像两个数组元素那样) 两个共享变量, thread_1 访问的共享变量仅有 x , thread_2 访问的共享变量仅有 y 。由于 x 、 y 在内存地址上是相邻的, 不妨进一步假设这两个变量被处理器加载到同一缓存条目的缓存行中。当 thread_1 更新 x 的时候, Processor 0 会发出一个 Invalidate 消息, 从而使其他处理器 (Processor 1) 上包含 x 的副本的缓存条目被无效化。此后, thread_2 访问 y 的时候, 由于包含这个变量副本的缓存条目已经因为此前 Processor 0 发出的 Invalidate 消息而被无效化了, 因此 thread_2 这时会遇到一个缓存未命中。

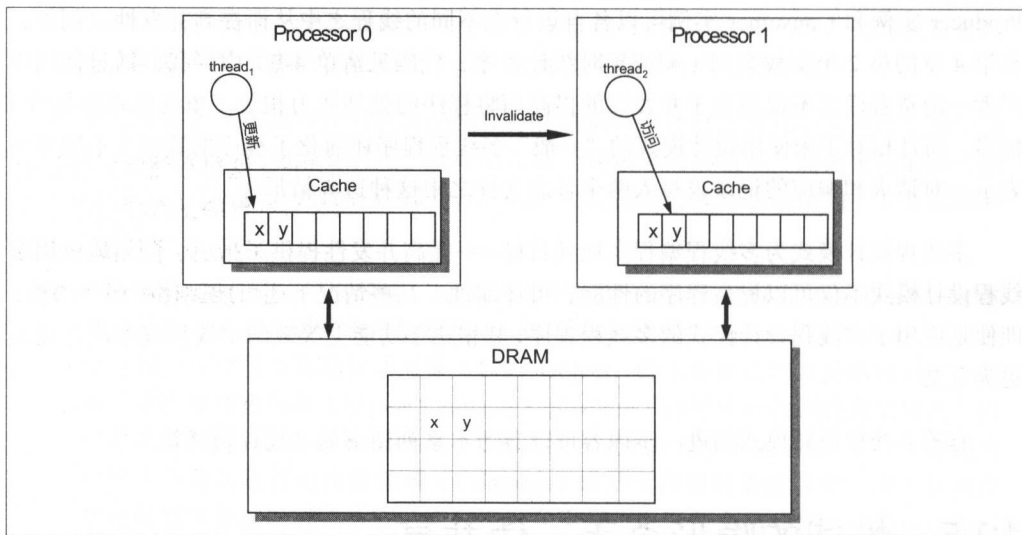


图 12-6 伪共享示意图

伪共享会导致缓存未命中, 从而降低处理器执行内存读、写操作的效率。

那么我们如何确定两个共享变量在内存地址上是否是相邻或者说这两个变量是否会被加载到同一个缓存行之中呢? 目前还没有有效的工具能够确认这一点。但是, 了解 Java 对象在内存中是如何存储的, 即 Java 对象的内存布局问题有助于我们分析与推断两个共享变量是否可能被加载到同一个缓存行之中。

12.5.1 Java 对象内存布局

我们知道 Java 对象是存储在堆内存 (Heap Memory) 之中的, 但是一个 Java 对象在

堆内存之中具体又是如何存储的呢？这就是 Java 对象的内存布局 (Memory Layout) 问题。对象的内存布局就是指对象在内存中的存储是如何组织的，或者说以什么样的形式在内存中呈现。由于同一个类的实例在不同的 Java 虚拟机下可能有着不同的内存布局，因此下面的讨论是基于 Oracle Hotspot 虚拟机的。

Java 对象在内存中的存储包括对象头 (Object Header) 和实例字段。其中，对象头会使用 2 个字 (Word) 的存储空间：第 1 个字用于存储对象的 HashCode、锁的相关信息 (比如偏向锁的偏向线程的 ID) 等信息；第 2 个字用于存储对象所属类的指针。因此对象头会占用 8 字节 (32 位处理器下) 或者 16 字节 (64 位处理器下)。另外，如果对象是一个数组，那么 Java 虚拟机会使用额外的一个字来表示数组的长度，即数组的对象头会占用 3 个字的空间。为了节约空间，在 4 字节足以表示对象所属类的地址的情况下，Java 虚拟机会仅使用 4 字节来表示对象头中的第 2 个字。因此，在 64 位系统下对象头可能只占用 12 (8+4) 字节。

总的来说，Java 虚拟机会为待创建的对象分配一段存储空间。这段存储空间的起始位置处 (位置偏移为 0) 存储的是对象头，对象头占用的空间之后存储的是对象的各个实例字段。当然，实例字段的数量可能是 0 (无状态对象) 也可能是多个。为了提高内存访问的效率并减少由此导致的内存空间的浪费，Java 虚拟机会依照一定的规则将一个对象的对象头及该对象包含的实例字段分配到内存空间中进行存储¹⁵。

规则 1 对象是以 8 字节为粒度 (Granularity) 进行对齐 (Aligned) 的。

这个规则也被称为对象是 8 字节对齐 (8-Bytes Aligned) 进行存储的。所谓的 8 字节对齐，可以这样理解：把内存空间看成一个一个“小格子”，其中每个小格子的容量是 8 字节。如图 12-7 所示，假设我们有 4 个数据 a (占用 4 字节)、b (占用 4 字节)、c (占用 4 字节) 和 d (占用 8 字节) 要存入内存，并且这些数据要按照它们所占用的空间大小的顺序进行存储¹⁶。那么，我们可以先将 a 存入第 1 个小格子中，此时这个小格子还剩余的 4 字节的空间可以用来存放 b。然后，我们将 c 存入第 2 个小格子。这时我们只剩下 d 待存储，但是此时第 2 个小格子还剩的 4 字节的空间无法用来存储需要 8 字节空间的 d。因此，我们先将第 2 个小格子填满，填满的方法就是往其中存储一个占用 4 字节的填充材料 (Padding)。接着，我们便可以将 d 存入第 3 个格子。

15 具体参见：<http://www.programering.com/a/MDO2YjMwATE.html>。

16 这个假设只是为了便于讲解对齐的含义，与 Java 虚拟机的实际内存布局无必然联系。

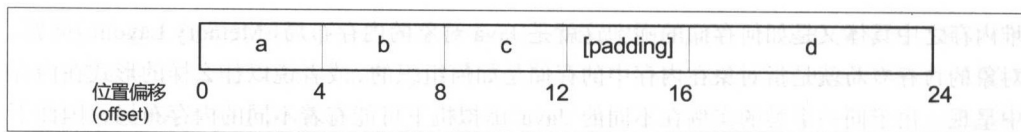


图 12-7 8 字节对齐数据存储示意图

可见，所谓“对齐”往往需要“填充”，而“填充”会带来一定的存储空间浪费。为了尽量减少这种浪费，Java 虚拟机在存储实例变量的时候并不是依照源代码中的声明顺序而是依照实例变量所占用的空间大小顺序进行。在 Java 平台中，boolean/byte 型变量占用 1 字节的空间，short/char 型变量占用 2 字节的空间，int/float 型变量占用 4 字节的空间，long/double 型变量占用 8 字节的空间，引用型变量占用 4 字节（32 位系统）或者 8 字节（64 位系统，且不开启指针压缩）的空间。由此，我们引出另外一条规则。

规则 2 对象中的实例字段按照如下顺序而非其源代码声明顺序排列。

- ① long 型变量和 double 型变量
- ② int 型变量和 float 型变量
- ③ short 型变量和 char 型变量
- ④ boolean 型变量和 byte 型变量
- ⑤ 引用型变量

规则 2 在某些情形下出于节约空间的需要也可能被打破。

与类的继承层次有关的规则是规则 3。

规则 3 继承自父类的实例字段不会与类本身定义的实例字段混杂在一起进行存储。

还有一些规则限于篇幅这里就不展开了。OpenJDK 项目下有个工具 jol（Java Object Layout）可以用来查看对象的实际内存布局¹⁷。例如，使用如下命令可以查看 ThreadLocalRandom 的对象布局：

```
java -XX:-RestrictContended -cp ./jol/jol-cli-0.6-full.jar org.openjdk.jol.Main
internals java.util.concurrent.ThreadLocalRandom
```

该命令输出如下：

17 下载地址：<http://central.maven.org/maven2/org/openjdk/jol/jol-cli/0.6/jol-cli-0.6-full.jar>。

```
# Running 64-bit HotSpot VM.
# Using compressed oop with 0-bit shift.
# Using compressed klass with 3-bit shift.
# Objects are 8 bytes aligned.
# Field sizes by type: 4, 1, 1, 2, 2, 4, 4, 8, 8 [bytes]
# Array element sizes: 4, 1, 1, 2, 2, 4, 4, 8, 8 [bytes]

java.util.concurrent.ThreadLocalRandom object internals:
OFFSET  SIZE        TYPE DESCRIPTION           VALUE
   0     4          (object header)          【省略此处输出】
   4     4          (object header)          【省略此处输出】
   8     4          (object header)          【省略此处输出】
  12     1  boolean Random.haveNextNextGaussian  false
  13     3          (alignment/padding gap)      N/A
  16     8    double Random.nextNextGaussian      0.0
  24     4  AtomicLong Random.seed              (object)
  28     1    boolean ThreadLocalRandom.initialized  true
  29     3          (loss due to the next object alignment)

Instance size: 32 bytes
Space losses: 3 bytes internal + 3 bytes external = 6 bytes total
```

从这个输出中可以看出，对象所占用的存储空间里先存储的是对象头，接下来才是对象的各个实例字段。另外，这个输出里的规则 2 被打破了。

12.5.2 伪共享的侦测与消除

下面我们通过一个伪共享 Demo 来介绍伪共享的侦测与消除。如清单 12-11 所示，我们连续创建了 N 个 CountingTask 接口实例，并相应地创建 N 个工作者线程（FalseSharingDemo 实例）。这些工作者线程通过其实例变量 task 访问各自的 CountingTask 实例，相互之间并不存在共同的共享变量，因此这里的实例变量 task 实际上是这些工作者线程的线程特有对象。

清单 12-11 伪共享 Demo

```
/**
 * 伪共享 Demo
 *
 * @author Viscent Huang
 */
public class FalseSharingDemo extends Thread {
    final CountingTask task;

    public FalseSharingDemo(CountingTask task) {
        this.task = task;
    }

    @Override
```

```

public void run() {
    final CountingTask t = task;
    final long count = t.getIterations();
    for (long i = 0; i < count; i++) {
        t.setValue(t.getValue() + i);
    }
}

public static void main(String[] args) throws Exception {
    int argc = args.length;
    int N; // 工作者线程数
    N = argc > 0 ? Integer.valueOf(args[0]) : Runtime.getRuntime()
        .availableProcessors();
    long iterations;
    iterations = argc > 1 ? Long.valueOf(args[1])
        : 400 * 1000 * 1000L;

    String taskImplClassName;
    taskImplClassName = System.getProperty("x.task.impl");
    if (null == taskImplClassName) {
        taskImplClassName = "DefaultCountingTask";
    }

    CountingTask[] tasks = createTasks(taskImplClassName, N, iterations);
    Thread[] demoThreads = new Thread[N];
    for (int i = 0; i < N; i++) {
        demoThreads[i] = new FalseSharingDemo(tasks[i]);
    }
    long start = System.currentTimeMillis();
    // 启动并等待指定的线程终止
    Tools.startAndWaitTerminated(demoThreads);
    System.out
        .printf("Duration: %d ms %n", System.currentTimeMillis() - start);
}

private static CountingTask[] createTasks(String taskImplClassName, int N,
    long iterations) {
    CountingTask[] tasks = new CountingTask[N];
    // 这里必须连续创建多个 XXCountingTask 实例,
    // 创建这些实例期间不能创建其他实例以提高 Java 虚拟机为这些对象分配连续的内存空间的概率。
    if ("DefaultCountingTask".equals(taskImplClassName)) {
        for (int i = 0; i < N; i++) {
            tasks[i] = new DefaultCountingTask(iterations);
        }
    } else if ("AutoPaddedCountingTask".equals(taskImplClassName)) {
        for (int i = 0; i < N; i++) {
            tasks[i] = new AutoPaddedCountingTask(iterations);
        }
    } else {
        for (int i = 0; i < N; i++) {

```

```

        tasks[i] = new ManuallyPaddedCountingTask(iterations);
    }
}
return tasks;
}
}

```

FalseSharingDemo 默认情况下创建的 CountingTask 实例是 DefaultCountingTask (源码见清单 12-12) 实例。

清单 12-12 DefaultCountingTask 源码

```

public class DefaultCountingTask implements CountingTask {
    private final long iterations;
    private volatile long value;

    public DefaultCountingTask() {
        this(100_0000);
    }

    public DefaultCountingTask(long iterations) {
        this.iterations = iterations;
    }

    @Override
    public long getIterations() {
        return iterations;
    }

    @Override
    public void setValue(long value) {
        this.value = value;
    }

    @Override
    public long getValue() {
        return value;
    }
}

```

通过指定不同的工作者线程数，我们可以看到如表 12-1 所示的运行结果¹⁸。可见，随着工作者线程数的增加，所有工作者线程的总耗时也随之增加。由于该 Demo 中的工作者

18 运行环境配置——操作系统：64 位 Linux；处理器：两个双核 CPU，2.50GHz；高速缓存：3 级高速缓存，一级缓存容量为 64KB（其中，数据缓存与指令缓存各占 32KB），缓存行宽度为 64 字节；JDK：JDK 1.8.0_40，64 位。

线程的任务处理逻辑非常简单——在循环中进行简单的字段读取和更新，并且各个工作者线程间也不存在共同的共享变量；因此，只要指定的工作者线程数量不超过系统的总处理器数目，那么这些工作者线程就应该可以并行（或者几乎并行）。也就是说这些工作者线程的总耗时应该仍然与一个工作者线程的情况相近，而实际则不然：此时工作者线程的总耗时却是原来的 6.7（=19377/2880）倍之多！

表 12-1 FalseSharingDemo 运行耗时

工作者线程数	所有工作者线程的总耗时（毫秒）
1	2 880
2	3 078
3	14 229
4	19 377

使用如下两个命令比较工作者线程数为 1 和 4 的情形下（单线程和多线程）的性能指标，如表 12-2 所示。

```
perf stat -e cpu-clock,task-clock,cs,instructions,L1-dcache-load-misses,\
L1-dcache-store-misses,LLC-loads,LLC-stores \
java io.github.viscent.mtia.ch12.FalseSharingDemo 1

perf stat -e cpu-clock,task-clock,cs,instructions,L1-dcache-load-misses,\
L1-dcache-store-misses,LLC-loads,LLC-stores \
java io.github.viscent.mtia.ch12.FalseSharingDemo 4
```

表 12-2 FalseSharingDemo 单线程和多线程执行的性能指标对比

工作者线程数	性能指标
1	2971.255072 cpu-clock (msec)
	2971.261197 task-clock (msec) # 1.004 CPUs utilized
	524 cs # 0.176 K/sec
	3,817,391,370 instructions
	2,838,882 L1-dcache-load-misses # 0.955 M/sec
	1,288,937 L1-dcache-store-misses # 0.434 M/sec
	1,010,659 LLC-loads # 0.340 M/sec
	449,098 LLC-stores # 0.151 M/sec
	2.960300071 seconds time elapsed
	51250.195935 cpu-clock (msec)
4	51250.104684 task-clock (msec) # 2.633 CPUs utilized
	5,370 cs # 0.105 K/sec

续表

工作者线程数	性能指标		
4	11,107,828,987 instructions		
	1,129,034,813 L1-dcache-load-misses	#	22.030 M/sec
	15,752,960 L1-dcache-store-misses	#	0.307 M/sec
	377,163,374 LLC-loads	#	7.359 M/sec
	743,452,840 LLC-stores	#	14.506 M/sec
	19.465587054 seconds time elapsed		

从表 12-2 中可以看出，工作者线程数为 4 的情况下，程序运行过程中消耗的总处理器时间（cpu-clock）约为 51 秒（51 250 毫秒）。如果这 4 个工作者线程接近并行的话，那么程序运行的总耗时应该是 13 秒左右（ $51/4=12.75$ ），而实际的程序总耗时却是 19 秒之多！这当中多出 6 秒的时间不可能仅仅是由于额外的（ $4846=5370-524$ ）上下文切换造成的，因此导致这个程序缓慢的因素应该是其他的资源竞争带来的等待。与工作者线程数为 1 的情形相比，我们注意到此时不仅上下文切换（cs）增加了（意料之中的），一级数据缓存（L1d）上的读缓存未命中（L1-dcache-load-misses）的数量也急剧增加。因此，此时程序缓慢极有可能是伪共享导致的缓存未命中的剧增从而增加了内存访问的延时而导致的。我们通过使用 jol 查看 DefaultCountingTask 实例的内存布局来确认这个合理的假设，如下所示：

```
# Running 64-bit HotSpot VM.
# Using compressed oop with 0-bit shift.
# Using compressed klass with 3-bit shift.
# Objects are 8 bytes aligned.
# Field sizes by type: 4, 1, 1, 2, 2, 4, 4, 8, 8 [bytes]
# Array element sizes: 4, 1, 1, 2, 2, 4, 4, 8, 8 [bytes]

io.github.viscent.mtia.ch12.DefaultCountingTask object internals:
OFFSET  SIZE  TYPE  DESCRIPTION  VALUE
   0     4      (object header)      【省略此处输出】
   4     4      (object header)      【省略此处输出】
   8     4      (object header)      【省略此处输出】
  12     4      (alignment/padding gap)  N/A
  16     8  long  DefaultCountingTask.iterations 1000000
  24     8  long  DefaultCountingTask.value      0

Instance size: 32 bytes
Space losses: 4 bytes internal + 0 bytes external = 4 bytes total
```

可见，一个 DefaultCountingTask 实例占用 32 字节的内存空间。因此两个 DefaultCountingTask 实例正好是可以被加载到一个宽度为 64 字节的缓存行之中。再加上我们在程序中连续创建多个 DefaultCountingTask 实例，因此连续创建的两个 DefaultCountingTask 实例极有可

能被 Java 虚拟机安排在连续的内存空间中进行存储。由此，我们基本上可以断定上述程序缓慢的问题是伪共享导致的。

消除伪共享的一个方法就是填充（Padding）。由于伪共享产生的前提是多个线程访问了位于同一缓存行之中的共享变量（尽管这些线程并没有访问同一个共享变量），因此消除伪共享的一个直观思路就是设法不让这些线程所访问的共享变量被加载到同一个缓存行之中。填充就是通过添加一些“无用的”（没有功能上的用途）实例变量来“干扰”对象的内存布局，以使特定的实例变量（或者某个实例）能够独自占用一个缓存行的空间，从而避免这些实例变量（或者实例）与其他实例变量（或者实例）被加载到同一个缓存行之中。

从 DefaultCountingTask 实例的内存布局来看，该实例在 value 实例变量前占用了 24 字节的空间。因此，如果两个 DefaultCountingTask 实例被分配到连续的内存空间中进行存储，那么这两个实例的 value 实例变量间的“距离”（两个变量相对于各自对象的位置偏移之差）就是 24 字节。于是，假设高速缓存的缓存行宽度为 64 字节，我们可以在 DefaultCountingTask 类的 value 实例变量之后填充 40（=64-24）字节的内容，从而使得任意两个 DefaultCountingTask 实例的 value 字段之间的“距离”均超过缓存行宽度而无法被“放入”同一个缓存行之中。根据这个思路，我们可以创建 CountingTask 新的实现类 ManuallyPaddedCountingTask，如清单 12-13 所示。

清单 12-13 ManuallyPaddedCountingTask 源码

```
public class ManuallyPaddedCountingTask implements CountingTask {
    private final long iterations;
    public volatile long value;
    // 填充
    protected volatile long p1, p2, p3, p4;

    // .....
}
```

为谨慎起见，我们还是再次使用 jol 确认一下 ManuallyPaddedCountingTask 实例的内存布局是否符合我们的期望：

```
# Running 64-bit HotSpot VM.
# Using compressed oop with 0-bit shift.
# Using compressed klass with 3-bit shift.
# Objects are 8 bytes aligned.
# Field sizes by type: 4, 1, 1, 2, 2, 4, 4, 8, 8 [bytes]
# Array element sizes: 4, 1, 1, 2, 2, 4, 4, 8, 8 [bytes]
```

io.github.viscent.mtia.ch12.ManuallyPaddedCountingTask object internals:

OFFSET	SIZE	TYPE	DESCRIPTION	VALUE
0	4		(object header)	【省略此处输出】
4	4		(object header)	【省略此处输出】
8	4		(object header)	【省略此处输出】
12	4		(alignment/padding gap)	N/A
16	8	long	ManuallyPaddedCountingTask.iterations	1000000
24	8	long	ManuallyPaddedCountingTask.value	0
32	8	long	ManuallyPaddedCountingTask.p1	0
40	8	long	ManuallyPaddedCountingTask.p2	0
48	8	long	ManuallyPaddedCountingTask.p3	0
56	8	long	ManuallyPaddedCountingTask.p4	0

Instance size: 64 bytes

Space losses: 4 bytes internal + 0 bytes external = 4 bytes total

可见, ManuallyPaddedCountingTask 的 value 实例变量后面多了 32 ($=4 \times 8$) 字节的填充空间, 而整个 ManuallyPaddedCountingTask 实例会占用 64 字节。此时, 由于两个 ManuallyPaddedCountingTask 实例无法被加载到一个宽度为 64 字节的缓存行之中, 因此不同 ManuallyPaddedCountingTask 实例的 value 实例变量就不可能位于同一个缓存行, 从而消除了伪共享产生的前提。

使用如下命令将 ManuallyPaddedCountingTask 指定为 CountingTask 接口实现类再次运行本 Demo。

```
perf stat -e cpu-clock,task-clock,cs,instructions,L1-dcache-load-misses,\
L1-dcache-store-misses,LLC-loads,LLC-stores \
java -Dx.task.impl=ManuallyPaddedCountingTask \
io.github.viscent.mtia.ch12.FalseSharingDemo 4
```

可以看到类似如下的输出:

Duration: 6,534 ms

```
25954.395625 cpu-clock (msec)
25954.343466 task-clock (msec)      #    3.925 CPUs utilized
      3,791 cs                      #    0.146 K/sec
14,679,012,826 instructions
  4,813,297 L1-dcache-load-misses    #    0.185 M/sec
  1,797,457 L1-dcache-store-misses   #    0.069 M/sec
  1,700,754 LLC-loads                 #    0.066 M/sec
  1,045,644 LLC-stores                #    0.040 M/sec
```

6.613258160 seconds time elapsed

这个输出与表 12-2 中 4 个工作者线程时的输出相比, 一级数据缓存上的读缓存未命

中（L1-dcache-load-misses）呈现出数量级上的减少，所有工作者线程的总耗时也降到原来（未使用填充技术前）的 1/3 左右。可见，使用 DefaultCountingTask 作为 CountingTask 接口实现类时程序运行出现的缓慢现象的确是由伪共享造成的。另外，此时程序运行过程中消耗的总处理器时间约为 26 秒（25 954 毫秒），这样算起来每个工作者线程的平均处理器消耗为 6.5 秒（ $6.5=26/4$ ），而这个数字与整个程序的耗时 6.6（秒）非常接近。可见，消除伪共享之后的 4 个工作者线程几乎是并行的，从而使得整个程序的执行效率得以提升 3 倍之多！而实际上我们所做的仅仅是增加一些“无用的”字段，并没有更改程序的算法！

与其他多线程有关的问题类似，伪共享问题由于与缓存行宽度以及对象的具体内存布局有关，因此也不是必然出现的。

填充虽然能够在不改变程序算法的情况下使得程序的性能有显著的提升，但是，填充显然是一种以空间换时间的优化手段，因此大规模地使用填充可能导致过多的额外空间消耗，从而增加垃圾回收器的负担。另外，要正确地实现填充，我们必须需要知道系统的缓存行宽度，还要了解和确定 Java 对象的内存布局。然而，Java 语言本身并没有提供用于获取缓存行宽度的接口，并且不同处理器的缓存行宽度也可能不一样（从 16 字节到 128 字节不等）。因此，对缓存行宽度的依赖使得填充这种技术存在硬件层面的可移植性问题（更换或者升级机器）。填充时具体在字段声明的什么地方、填充多少个字节的内容实际上取决于对象的内存布局。而要了解 Java 对象的内存布局无疑增加了对人员的要求，并且由于不同的 Java 虚拟机可能有不同的内存布局规则，因此，对 Java 对象内存布局的依赖同样也使得填充这种技术存在软件层面的可移植性问题（部署在 Java 虚拟机上，或者切换 Java 虚拟机）。

直接在 Java 源代码这一层实现的填充（如清单 12-13 所示）被称为手动填充。手动填充不仅存在上述几个问题，而且还需要注意：首先，Java 虚拟机可能会将“无用的”字段给优化掉。因此，在清单 12-13 中我们填充的 4 个字段，尽管其所在类以及其他类都不需要访问这些字段，但是我们却使用 protected（而不是 private）和 volatile 来修饰这些变量以达到“欺骗”Java 虚拟机的目的——这些变量是有可能被当前线程或者其他线程访问的，否则 Java 虚拟机就会认为这些字段是“无用的”而将其优化掉。另外，如果要填充比较大的空间，比如需要填充 128 字节而不是像清单 12-13 那样仅需要填充 32 字节，那么我们需要用“protected volatile long[] paddings=new long[128]”这样的填充方式。

Java 8 根据 JDK 第 142 号增强提案（JEP142，<http://openjdk.java.net/jeps/142>）引入了一个特殊的注解 @sun.misc.Contended，该注解可以用来注释字段和类¹⁹。@sun.misc.

¹⁹ @sun.misc.Contended 并不对静态字段起作用。

Contended 的作用是给 Java 虚拟机一个提示——被注释的字段（仅限实例变量）或者类的实例可能面临伪共享问题。Java 虚拟机则根据这个注解进行填充来使得被注释的实例变量或者类的实例能够被加载到单独的一个缓存行之中。因此，这种填充被称为自动填充。使用自动填充，我们可以创建 CountingTask 接口的另外一个实现类 AutoPaddedCountingTask（见清单 12-14）来消除伪共享。

清单 12-14 AutoPaddedCountingTask 源码

```
public class AutoPaddedCountingTask implements CountingTask {
    private final long iterations;

    @sun.misc.Contended
    public volatile long value;
    // .....
}
```

由于目前默认情况下 @sun.misc.Contended 仅开放给 JDK 内部的类，因此，应用自身的类要使用该注解时需要开启 Java 虚拟机的开关“-XX:-RestrictContended”。所以，要使用如清单 12-14 所示的 CountingTask 实现类来消除伪共享，我们需要使用如下命令来运行本 Demo：

```
java -XX:-RestrictContended -Dx.task.impl=AutoPaddedCountingTask
io.github.viscent.mtia.ch12.FalseSharingDemo
```

自动填充避免了手动填充存在的一些问题（可移植性问题）和不便，但是它比手动填充更耗空间——JDK 1.8 在 @sun.misc.Contended 注释的字段（或者类的实例）前和后各自填充大小为缓存行宽度的 2 倍的填充空间。因此，依照性能优化“避免过早优化”的原则，我们应该只在确认存在伪共享问题的情况下才考虑使用填充。

减少共享变量的访问频率有助于降低伪共享问题出现的频率。例如，针对本 Demo 中的工作者线程的 run 方法（见清单 12-12），我们可以在不改变程序语义的前提下通过优先使用局部变量来减少共享变量的访问频率，从而降低伪共享问题出现的频率，如下代码片段所示：

```
public void run() {
    final CountingTask t = task;
    final long count = t.getIterations();
    long sum = 0;
    for (long i = 0; i < count; i++) {
        sum += i;
    }
    // 仅访问一次共享变量
    t.setValue(sum);
}
```

经过上面的调整本 Demo 的伪共享问题几乎不存在，并且 `volatile` 变量（`CountingTask` 中的 `value` 实例变量）访问的开销也极大地被降低了，因此程序的运行持续时间可以降低到几百毫秒。

虽然降低共享变量的访问频率所带来的效果可能比较明显，但是由于它可能涉及程序算法的调整，因此其运用比较受限。

12.6 本章小结

本章介绍了与 Java 多线程程序紧密相关的性能调校常用技术。本章知识结构如图 12-8 所示。

Java 虚拟机自 Java 6 开始对内部锁进行了若干优化：锁消除、锁粗化、偏向锁以及适应性锁。除锁消除是 Java 7 开始引入的，其他优化均是在 Java 6 开始引入的，这些优化仅在 Java 虚拟机的 `server` 模式下起作用。这些优化默认都是开启的，且多数优化都可能依赖于 JIT 的内联优化，并且其本身也可能是通过 JIT 编译实现的。因此，这些优化都有其开销。锁消除优化能够彻底消除锁的开销，它依赖于逃逸分析技术。锁粗化优化能够减少线程申请/释放锁的频率，其代价是使临界区长度变大，从而可能导致线程在申请锁时的等待时间变长。偏向锁优化可以减小锁的申请/释放的开销，它不适用于争用程度较高的锁。适应性锁优化可以减小锁申请的开销，有利于减少上下文切换。

锁的开销主要是由争用锁引起的。这些开销主要包括：上下文切换与线程调度开销、内存同步、编译器优化受限的开销以及限制可伸缩性。降低锁的开销可以从使用锁的替代品、降低锁的争用程度以及减少线程所需申请的锁的数量这几个方面入手。

使用可参数化锁可以减少线程所需申请的锁的数量从而降低锁的开销，但是它在一定程度上破坏了封装性。

减小临界区的长度可以减少锁的持有时间，从而降低锁的争用程度。减小临界区的长度有利于适用性锁优化发挥作用。在不影响线程安全的前提下，将临界区中的阻塞式 I/O 等阻塞操作以及较耗时的操作挪动到临界区之外可以减小临界区的长度。

减小锁的粒度可以降低锁的申请频率从而降低锁的争用程度。减小锁的粒度常用技术包括锁拆分技术和锁分段技术。锁拆分技术在高争用情况下的效果可能并不明显；锁分段技术会使得对整个对象进行加锁比较困难乃至不可能。

减少上下文切换可以从这几个方面入手：控制线程数量、避免在临界区中执行阻塞式 I/O 等阻塞操作、避免在临界区中执行比较耗时的操作和减少 Java 虚拟机垃圾回收。

运用多线程设计模式也有助于提升多线程程序的性能，但是程序的复杂性也可能相应增加。

伪共享产生的前提是多个线程访问被缓存到同一个缓存中的不同变量，它会导致大量的缓存未命中，从而增加内存访问操作的开销。了解 Java 对象的内存布局有助于分析与消除伪共享。Java 对象内存布局的规则包括：对象是以 8 字节为粒度（Granularity）进行对齐的、对象中的实例字段并非依照其源代码声明顺序排列以及继承自父类的实例字段不会与类本身定义的实例字段混杂在一起进行存储等。使用 jol 工具可以查看具体对象的内存布局情况。判断伪共享是否存在可以从分析多个线程是否存在共同的共享变量入手，并通过 jol 以及 Linux 内核工具 perf 来进一步分析与确认。伪共享可通过手工填充、自动填充以及降低共享变量的访问频率这几个方面来消除与规避。手工填充和自动填充可以在无须调整程序算法的前提下消除伪共享。手工填充的缺点比较多，使用该方法我们必须知道缓存行的宽度、Java 对象的具体内存布局，这使得该方法存在硬件、软件层面的可移植性问题，并对人员的要求比较高。并且，我们还需要避免手工填充的填充字段被 Java 虚拟机优化掉。自动填充依赖于@Contended 注解，它避免了手动填充的缺点，但是其消耗的额外空间更多。Java 虚拟机对自动填充的支持需要通过 Java 虚拟机的开关“-XX:-RestrictContended”开启。虽然减少共享变量的访问频率所带来的效果可能比较明显，但是由于它可能涉及程序算法的调整，因此其适用范围比较有限。

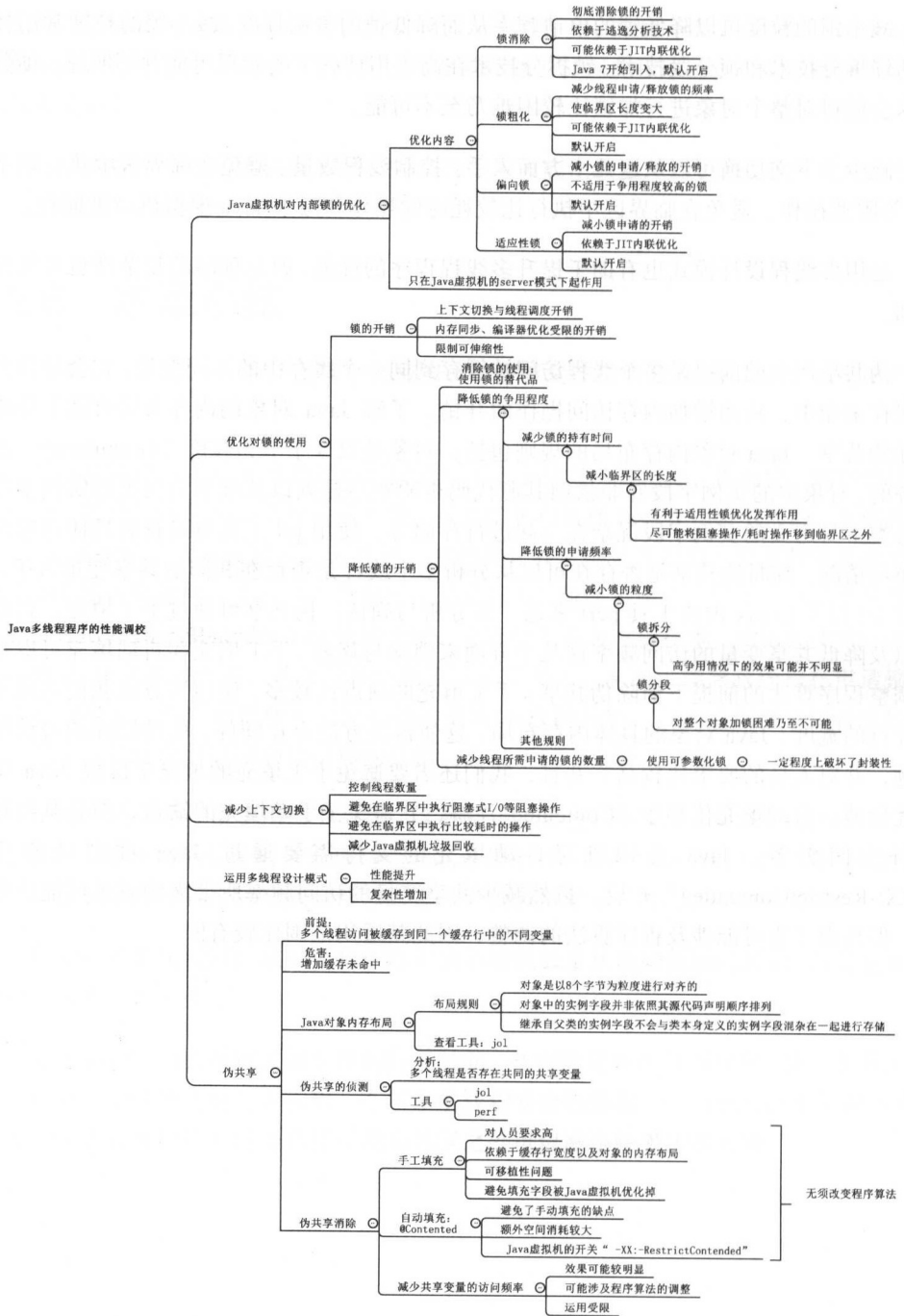


图 12-8 本章知识结构图

第 1 章 走近 Java 世界中的线程

1.1 Thread 类的被废弃（Deprecated）方法的废弃原因及替代方案

<http://docs.oracle.com/javase/7/docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html>

1.2 如何分析 Java 线程转储

<https://dzone.com/articles/how-analyze-java-thread-dumps>

1.3 jvisualvm 文档

<http://docs.oracle.com/javase/7/docs/technotes/guides/visualvm/index.html>

1.4 Java Mission Control（JMC）Eclipse 插件

<http://download.oracle.com/technology/products/missioncontrol/updatesites/base/5.2.0/eclipse/>

第 2 章 多线程编程的目标与挑战

2.1 循环不变表达式优化（Loop-invariant code motion）

<http://www.compileroptimizations.com/category/hoisting.htm>

2.2 Java 语言规范第 17 章

<http://docs.oracle.com/javase/specs/jls/se8/html/jls-17.html#jls-17.7>

2.3 Java 虚拟机的 client 模式与 server 模式的区别

http://www.oracle.com/technetwork/java/hotspotfaq-138619.html#compiler_types

2.4 perf 工具手册

https://perf.wiki.kernel.org/index.php/Main_Page

2.5 The JSR-133 Cookbook for Compiler Writers

<http://gee.cs.oswego.edu/dl/jmm/cookbook.html>

2.6 共享内存模型教程（Shared Memory Consistency Models: A Tutorial）

<http://rsim.cs.uiuc.edu/~sadve/Publications/computer96.pdf>

2.7 内存重排序

https://en.wikipedia.org/wiki/Memory_ordering

第 3 章 Java 线程同步机制

3.1 Java 虚拟机对内部锁的调度

<http://www.artima.com/insidejvm/ed2/threadsynch.html>

3.2 内存屏障与 Java 线程同步机制的实现（The JSR-133 Cookbook for Compiler Writers）

<http://gee.cs.oswego.edu/dl/jmm/cookbook.html>

3.3 Java 中的对象安全发布与初始化（Safe Publication and Safe Initialization in Java）

<https://shipilev.net/blog/2014/safe-public-construction/>

第 5 章 线程间协作

5.1 Java 多线程编程模式实战指南（三）：Two-phase Termination 模式

<http://www.infoq.com/cn/articles/java-multithreaded-programming-mode-two-phase-termination>

5.2 Disruptor 框架

<https://lmax-exchange.github.io/disruptor>

第 6 章 保障线程安全的设计技术

6.1 Java 存储空间

http://blog.jamesdbloom.com/JVMInternals.html#jvm_system_threads

6.2 对象存储空间的分配

<http://www.ibm.com/developerworks/java/library/j-jtp09275/>

6.3 Java 垃圾回收

https://www.infoq.com/articles/Java_Garbage_Collection_Distilled

6.4 不可变对象对垃圾回收的益处

<http://www.ibm.com/developerworks/library/j-jtp01274/>

6.5 Java 多线程编程模式实战指南（二）：Immutable Object 模式

<http://www.infoq.com/cn/articles/java-multithreaded-programming-mode-immutable-object>

6.6 Tomcat 的内存泄漏检测功能

<https://wiki.apache.org/tomcat/MemoryLeakProtection#customThreadLocal>

6.7 并发集合类

<http://www.ibm.com/developerworks/library/j-jtp07233/>

第 7 章 线程的活性故障

7.1 读写锁导致的饥饿

<http://www.javaspecialists.eu/archive/Issue165.html>

第 8 章 线程管理

8.1 Why ThreadGroup is being criticised?

<http://stackoverflow.com/questions/3265640/why-threadgroup-is-being-criticised>

8.2 工厂模式

https://en.wikipedia.org/wiki/Factory_method_pattern

8.3 Thread pools and work queues

<https://www.ibm.com/developerworks/library/j-jtp0730/>

第 9 章 Java 异步编程

9.1 异步 Servlet

<https://docs.oracle.com/javaee/7/tutorial/servlets012.htm>

9.2 Java 多线程编程模式实战指南之 Promise 模式

<http://www.infoq.com/cn/articles/design-patterns-promise>

9.3 Fork and Join: Java Can Excel at Painless Parallel Programming Too!

<http://www.oracle.com/technetwork/articles/java/fork-join-422606.html>

第 10 章 Java 多线程程序的调试与测试

10.1 FindBugs 的 Bug 模式（Bug pattern）

<http://findbugs.sourceforge.net/factSheet.html>

10.2 FindBugs 能够检查出的多线程相关（Multithreaded Correctness）的 Bug 列表

<http://findbugs.sourceforge.net/bugDescriptions.html>

10.3 FindBugs 的过滤文件（Filter File）

<http://findbugs.sourceforge.net/manual/filter.html>

10.4 jctestress

<http://openjdk.java.net/projects/code-tools/jctestress/>

第 11 章 多线程编程的硬件基础与 Java 内存模型

11.1 处理器高速缓存概述

https://en.wikipedia.org/wiki/CPU_cache

11.2 处理器高速缓存的内部实现

<http://courses.cs.washington.edu/courses/cse378/09wi/lectures/lec15.pdf>

11.3 内存地址与高速缓存地址之间的转换

<http://www.ccs.neu.edu/course/com3200/parent/NOTES/cache-basics.html>

<https://cseweb.ucsd.edu/classes/su07/cse141/cache-handout.pdf>

11.4 缓存一致性

<http://meseec.ce.rit.edu/551-projects/fall2010/1-3.pdf>

11.5 MESI 协议

<http://www.rdrop.com/users/paulmck/scalability/paper/whymb.2010.07.23a.pdf>

11.6 Memory Barriers a Hardware View for Software Hackers

<http://www.rdrop.com/users/paulmck/scalability/paper/whymb.2010.07.23a.pdf>

11.7 Fixing the Java Memory Model

<https://www.ibm.com/developerworks/library/j-jtp02244/>

<http://www.ibm.com/developerworks/library/j-jtp03304/>

11.8 JSR 133 (Java Memory Model) FAQ

<http://www.cs.umd.edu/~pugh/java/memoryModel/jsr-133-faq.html>

第 12 章 Java 多线程程序的性能调校

12.1 Java theory and practice: Synchronization optimizations in Mustang

<http://www.ibm.com/developerworks/library/j-jtp10185/>

12.2 Do Java 6 threading optimizations actually work?

<https://www.infoq.com/articles/java-threading-optimizations-p1>

12.3 Biased Locking in HotSpot

https://blogs.oracle.com/dave/entry/biased_locking_in_hotspot

12.4 Java SE 6 Performance White Paper

<http://www.oracle.com/technetwork/java/6-performance-137236.html>

12.5 Java HotSpot™ Virtual Machine Performance Enhancements

<http://docs.oracle.com/javase/7/docs/technotes/guides/vm/performance-enhancements-7.html>

12.6 减少锁的争用

<http://www.ibm.com/developerworks/library/j-threads2/>

12.7 Java Garbage Collection Distilled

https://www.infoq.com/articles/Java_Garbage_Collection_Distilled

12.8 JEP142

<http://openjdk.java.net/jeps/142>

参考文献

1. Brian Göetz et al. Java Concurrency In Practice. Addison Wesley, 2006.
2. Doug Lea. Concurrent Programming in Java: Design Principles and Patterns, Second Edition. Addison Wesley, 1999.
3. Maurice Herlihy. The Art of Multiprocessor Programming. Morgan Kaufmann, 2012.
4. Scott Oaks. Java Performance: The Definitive Guide. O'Reilly Media, Inc. , 2014.
5. 黄文海. Java 多线程编程实战指南（设计模式篇）. 电子工业出版社, 2015.
6. Bill Venners, Inside the Java Virtual Machine. McGraw-Hill, 1999.
7. Erich Gamma 等. 设计模式：可复用面向对象软件的基础（英文版）. 机械工业出版社, 2002.
8. Randal E. Bryant et al. Computer Systems: A Programmer's Perspective, Second Edition. Prentice Hall, 2010.
9. Joshua Bloch. Effective Java: Programming Language Guide. Addison Wesley, 2001.
10. Robert C. Martin. Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall, 2009.

十载耕耘 奠定专业地位

以书为证 彰显卓越品质

博文视点诚邀精锐作者加盟

《C++Primer (中文版) (第5版)》、《淘宝技术这十年》、《代码大全》、《Windows内核情景分析》、《加密与解密》、《编程之美》、《VC++深入详解》、《SEO实战密码》、《PPT演义》……

“圣经”级图书光耀夺目,被无数读者朋友奉为案头手册传世经典。

潘爱民、毛德操、张亚勤、张宏江、咎辉Zac、李刚、曹江华……

“明星”级作者济济一堂,他们的名字熠熠生辉,与IT业的蓬勃发展紧密相连。

十年的开拓、探索和励精图治,成就博古通今、文圆质方、视角独特、点石成金之计算机图书的风向标杆:博文视点。

“凤翱翔于千仞兮,非梧不栖”,博文视点欢迎更多才华横溢、锐意创新的作者朋友加盟,与大师并列于IT专业出版之巅。

英雄帖

江湖风云起,代有才人出。

IT界群雄并起,逐鹿中原。

博文视点诚邀天下技术英豪加入,

指点江山,激扬文字

传播信息技术,分享IT心得

· 专业的作者服务 ·

博文视点自成立以来一直专注于IT专业技术图书的出版,拥有丰富的与技术图书作者合作的经验,并参照IT技术图书的特点,打造了一支高效运转、富有服务意识的编辑出版团队。我们始终坚持:

善待作者——我们会把出版流程整理得清晰简明,为作者提供优厚的稿酬服务,解除作者的顾虑,安心写作,展现出最好的作品。

尊重作者——我们尊重每一位作者的技术实力和生活习惯,并会参照作者实际的工作、生活节奏,量身制定写作计划,确保合作顺利进行。

提升作者——我们打造精品图书,更要打造知名作者。博文视点致力于通过图书提升作者的个人品牌和技术影响力,为作者的事业开拓带来更多的机会。



联系我们

博文视点官网: <http://www.broadview.com.cn>

CSDN官方博客: <http://blog.csdn.net/broadview2006/>

投稿电话: 010-51260888 88254368

投稿邮箱: jsj@phei.com.cn



@博文视点Broadview



微信公众账号

博文视点Broadview



Java 多线程编程 实战指南

(核心篇)

随着现代处理器的生产工艺从提升处理器主频频率转向多核化,即在一块芯片上集成多个处理器内核(Core),多核处理器(Multicore Processor)离我们越来越近了——如今就连智能手机这样的消费类设备都已配备了4核乃至8核的处理器,更何况商用系统!在此背景下,以往靠单个处理器自身处理能力的提升所带来的软件计算性能提升的那种“免费午餐”已不复存在,这使得多线程编程在充分利用计算资源、提高软件服务质量方面扮演了越来越重要的角色。故而,掌握多线程编程技能对广大开发人员的重要性亦由此可见一斑。

“Java多线程编程实战系列”书籍是全面介绍Java多线程编程的实战书籍,本系列图书包含两本书,分别是设计模式篇和核心篇。本书(核心篇)以基本概念、原理与方法为主线,辅以丰富的实战案例和生活化实例,并从Java虚拟机、操作系统和硬件多个层次与角度出发,循序渐进、系统地介绍Java平台下的多线程编程核心技术及相关工具。而设计模式篇则采用Java语言和UML为描述语言,并结合作者多年工作经历的相关实战案例,介绍多线程环境下常用设计模式的来龙去脉:各个设计模式是什么样的及其典型的实际应用场景、实际应用时需要注意的事项以及各个模式的可复用代码实现。希望读者通过本系列图书能够在掌握多线程编程知识与技能的旅途中少走弯路。



本书源码下载地址:

<https://github.com/Viscent/javamtia>

<http://www.broadview.com.cn/31065>



博文视点Broadview



@博文视点Broadview

上架建议:程序设计>Java

ISBN 978-7-121-31065-2



9 787121 310652 >

定价: 89.00元



策划编辑:付睿 @Winnie说说

责任编辑:李云静

封面设计:李玲